Exact and Approximate Maximum Inner Product Search with LEMP (draft / author version)

Christina Teflioudi, Max Planck Institute for Computer Science
Rainer Gemulla, University of Mannheim

We study exact and approximate methods for maximum inner product search, a fundamental problem in a number of data mining and information retrieval tasks. We propose the LEMP framework, which supports both exact and approximate search with quality guarantees. At its heart, LEMP transforms a maximum inner product search problem over a large database of vectors into a number of smaller cosine similarity search problems. This transformation allows LEMP to prune large parts of the search space immediately and to select suitable search algorithms for each of the remaining problems individually. LEMP is able to leverage existing methods for cosine similarity search, but we also provide a number of novel search algorithms tailored to our setting. We conducted an extensive experimental study that provides insight into the performance of many state-of-the-art techniques—including LEMP—on multiple real-world datasets. We found that LEMP often was significantly faster or more accurate than alternative methods.

CCS Concepts:•**Information systems** → **Data access methods;** *Top-k retrieval in databases;*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: maximum inner product search (MIPS), indexing, top-k search, recommender systems

ACM Reference Format:
Christina Teflioudi and Rainer Gemulla, 2015. Exact or Approximate Maximum Inner Product Search with LEMP . *ACM Trans. Datab. Syst.* V, N, Article A (January 2015), 46 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. Introduction

We study exact and approximate methods for maximum inner product search (MIPS). Given a large database of real-valued vectors—called *probe vectors*—as well as a *query vector*, the MIPS problem is to find all probe vectors that have a large inner product with the query vector. MIPS is a fundamental problem in a number of data mining and information retrieval tasks, including finding good recommendations in recommender systems [Koren et al. 2009; Koenigstein et al. 2012], reasoning about extracted facts in open relation extraction [Riedel et al. 2013], multi-class or multi-label prediction with hundreds of thousands labels or classes [Dean et al. 2013], and object detection with deformable part models [Dean et al. 2013; Shrivastava and Li 2014a].

In this article, we consider multiple variants of the MIPS problem, which differ in what is considered a large inner product, whether the search is exact or approximate, and whether there is just one or multiple query vectors. We focus on settings in which the number of vectors is very large (order of millions), the vectors have medium dimensionality (say, 10–500), and the vectors vary in L2 norm. This is a common setting in many MIPS applications.

A simple way to solve the MIPS problem is to perform naive search, i.e., to compute the inner product between the query vector and all probe vectors and output all probe vectors with large inner products. Such an approach is generally computationally infeasible. Consider for example a MIPS problem with 10M probe and 10M query vectors of dimensionality 50. Then naive search conducts 100 trillion inner product computations. If each inner product computation takes about 100 ns on average (as in our experimental study), it takes more than 100 days for naive search to complete (ignoring any other costs such as I/O costs).

To avoid such expensive computations, a number of exact and approximate algorithms for MIPS have been proposed in the literature. These algorithms can generally be categorized into (i) methods customized for MIPS and (ii) methods that transform MIPS into a single similarity search problem. *Customized methods* aim to exploit properties of the MIPS problem in order to build suitable index structures. Examples include ball and cone tree methods [Ram and Gray 2012; Koenigstein et al. 2012], cover trees [Curtin et al. 2013; Curtin and Ram 2014], and the LEMP framework proposed in this article. *Transformation-based methods* are based on the observation that the MIPS problem can be reformulated as a similarity search problem in a (slightly) higher dimensional space. These methods transform a given MIPS instance into a single large cosine similarity search or Euclidean nearest-neighbor search problem, and then apply a suitable similarity search method. Examples of such approaches include PCA-Tree [Bachrach et al. 2014], asymmetric LSH [Shrivastava and Li 2014a; 2014b], and simpleLSH [Neyshabur and Srebro 2014]. Our LEMP framework differs from transformation-based methods because LEMP transforms a given MIPS instance into *multiple* search problems, instead of a single one.

In this article, we propose the LEMP framework[1] for both exact and approximate MIPS. LEMP makes use of the fact that both the L2 norms and the directions of two vectors influence the value of their inner product. LEMP exploits this fact by grouping the input vectors into *buckets* such that the vectors within a bucket have similar L2 norms. LEMP subsequently solves a smaller cosine similarity search problem for each bucket to obtain the final result. This procedure allows LEMP (i) to exploit the L2 norms of query and probe vectors for pruning buckets, (ii) to choose a suitable search technique individually for each remaining bucket (and query), and (iii) to improve cache locality by keeping buckets small so that they fit into the processor cache. LEMP supports approximate MIPS by using more aggressive pruning techniques as well as approximate methods for solving the within-bucket search problems. Our methods provide approximation guarantees and allow to trade-off result quality and speed.

To process buckets, LEMP is able to leverage any existing method for cosine similarity search or MIPS. We consider a number of such methods, including the well-known threshold algorithm (TA, [Fagin et al. 2001]) and techniques for cosine similarity search such as L2AP [Anastasiu and Karypis 2014] or cover trees [Curtin et al. 2013]. We propose two novel methods for exact cosine similarity search, termed COORD (for coordinate-based pruning) and ICOORD (for incremental coordinate-based pruning); our methods are tailored for their use within the LEMP framework and, according to our experimental study, are generally more efficient than alternative methods. We also propose four novel methods for approximate search: LEMP-ABS and LEMP-REL are based on ICOORD, LEMP-LSHA is based on an adaptive variant of locality-sensitive hashing (LSH, [Gionis et al. 1999]), and LEMP-HYB is a hybrid method.

We conducted an extensive experimental study, in which we compared state-of-the-art techniques—including LEMP—on multiple real-world datasets. Our study aims to

[1]A preliminary version of this article was published in [Teflioudi et al. 2015]. The name LEMP stems from finding Large Entries in a Matrix Product, a problem equivalent to MIPS.

provide insight into the relative performance of customized and transformation-based methods. For exact search, we found that LEMP consistently outperformed prior methods and can be multiple orders of magnitude faster than naive search. We also found that the performance of existing exact methods for MIPS increased when used within the LEMP framework; the best-performing method overall, however, was a variant of the ICOORD algorithm. For approximate MIPS, the best-performing method was dataset-dependent. We found that a combination of the symmetric transformation of Neyshabur and Srebro [2014] with the high-performance search algorithm Annoy[2] generally outperformed all prior methods but LEMP. On two out of the three datasets, LEMP offered the best trade-off between result quality and speed and was second only to Annoy on the third one. LEMP provided up to 2x higher recall (for a fixed execution time) and up to 3.7x speed-up (for a fixed quality level) compared to the best-performing previous method.

The remainder of this article is structured as follows: Sec. 2 discusses applications of MIPS and formally defines (multiple variants of) the MIPS problem. Sec. 3 introduces the LEMP framework. Sec. 4 and 5 focus on exact and approximate MIPS, respectively. Sec. 6 outlines how LEMP can be parallelized effectively. Sec. 7 summarizes related work. Sec. 8 describes our experimental study and its results. We conclude the article in Sec. 9.

2. Preliminaries and Problem Statement

In this section, we introduce the notation used throughout, describe applications of MIPS, and formally define multiple variants of the MIPS problem.

2.1. Notation

Let $[n] = \{1, \ldots, n\}$. We denote matrices by bold uppercase letters, vectors by bold lowercase letters, and scalars by non-bold lowercase letters. Throughout this article, we represent a set of vectors using a matrix in which each column holds one of the vectors. We write $\boldsymbol{M}_j$ for the j-th column of matrix $\boldsymbol{M}$, and $\boldsymbol{v} \in \boldsymbol{M}$ if $\boldsymbol{v}$ is a column of $\boldsymbol{M}$. We denote the i-th element of vector $\boldsymbol{v}$ by v_i and the L2 norm of $\boldsymbol{v}$ by $\|\boldsymbol{v}\| = \sqrt{\sum_i v_i^2}$.

Denote by $\boldsymbol{Q}$ and $\boldsymbol{P}$ two sets of r-dimensional vectors with cardinality m and n, respectively. We assume throughout that m and n are very large (order of millions) and r is comparably small (say, 10–500). We are interested in finding pairs of vectors, one from $\boldsymbol{Q}$ and one from $\boldsymbol{P}$, with large inner product or, equivalently, the indexes of the large entries in the product matrix $\boldsymbol{Q}^T\boldsymbol{P}$. We refer to $\boldsymbol{Q}$ as the *query matrix* and to $\boldsymbol{P}$ as the *probe matrix*. Similarly, we refer to vectors $\boldsymbol{q} \in \boldsymbol{Q}$ as *query vectors* (or simply *queries*) and to vectors $\boldsymbol{p} \in \boldsymbol{P}$ as *probe vectors*.

Fix a probe matrix $\boldsymbol{P}$. We refer to the inner product $\boldsymbol{q}^T\boldsymbol{p}$ as the *score* of $\boldsymbol{p} \in \boldsymbol{P}$ for query $\boldsymbol{q}$. For expository reasons, we assume throughout that all scores for $\boldsymbol{q}$ are distinct. Let $\boldsymbol{p}^{\boldsymbol{q}}_{(1)}, \ldots, \boldsymbol{p}^{\boldsymbol{q}}_{(n)}$ denote the probe vectors in $\boldsymbol{P}$ sorted by their score for $\boldsymbol{q}$ in descending order. We refer to $T_k(\boldsymbol{q}) = \{\boldsymbol{p}^{\boldsymbol{q}}_{(1)}, \ldots, \boldsymbol{p}^{\boldsymbol{q}}_{(k)}\}$ as the *top-k list* of $\boldsymbol{q}$ and to $s^{\boldsymbol{q}}_{(k)} = \boldsymbol{q}^T\boldsymbol{p}^{\boldsymbol{q}}_{(k)}$ as the *top-k score* of $\boldsymbol{q}$. When the query $\boldsymbol{q}$ is clear from context, we omit argument or superscript $\boldsymbol{q}$.

2.2. Applications of MIPS

The MIPS problem frequently arises in data mining tasks that employ some form of low-rank matrix factorization. Such low-rank matrix factorization methods—e.g., the singular value decomposition (SVD), non-negative matrix factorization (NMF),

[2]https://github.com/spotify/annoy

$$\begin{array}{l|ccccc} & \textit{Die Hard} & \textit{Taken} & \textit{Once} & \textit{Amelie} & \textit{Titanic} \\ \hline \textit{Anna} & 5 & & 1 & 2 & \\ \textit{Bob} & 5 & 4 & & & 1 \\ \textit{Charlie} & 2 & & 5 & & 4 \\ \textit{Debby} & & 1 & 5 & 5 & \end{array}$$

(a) Feedback matrix $\boldsymbol{D}$

$$\begin{pmatrix} 1.6 & 1.3 & 0.7 & 1 & 0.4 \\ 0.6 & 0.8 & 2.7 & 2.8 & 2.2 \end{pmatrix} \boldsymbol{P}$$

$$\underset{\boldsymbol{Q}^T}{\begin{pmatrix} 3.2 & -0.4 \\ 3.1 & -0.2 \\ 0 & 1.8 \\ -0.4 & 1.9 \end{pmatrix}} \quad \underset{\boldsymbol{Q}^T\boldsymbol{P}}{\begin{pmatrix} \mathbf{4.9} & \mathbf{3.8} & 1.2 & \mathbf{2.1} & 0.4 \\ \mathbf{4.8} & \mathbf{3.9} & 1.6 & \mathbf{2.5} & 0.8 \\ 1 & 1.4 & \mathbf{4.9} & \mathbf{5.0} & \mathbf{4.0} \\ 0.5 & 1 & \mathbf{4.9} & \mathbf{4.9} & \mathbf{4.0} \end{pmatrix}}$$

(b) Factor matrices for users ($\boldsymbol{Q}$) and movies ($\boldsymbol{P}$) as well as corresponding predictions ($\boldsymbol{Q}^T\boldsymbol{P}$)

Fig. 1: Example of a simple matrix factorization model for a recommender system

or latent-factor models—have been successfully applied to a number of prediction tasks [Skillicorn 2007]. In general, the available data is represented as a matrix in which rows and columns correspond to entities or attributes of interest, and entries to values. Low-rank matrix factorizations are used for dimensionality reduction and to reveal hidden structure in the data. Large entries in the obtained low-rank matrix indicate strong interactions between entities and attributes and are often of particular interest in applications.

In the context of recommender systems, for example, latent-factor models are a popular and powerful approach for predicting the preference of users for items from available feedback; see [Koren et al. 2009] for an excellent overview. Figure 1a shows a feedback matrix $\boldsymbol{D}$, which contains ratings of a set of users for a set of movies they had watched on a 1–5 star scale. To predict the ratings of the movies users did not yet see (or rate), latent-factor models construct two factor matrices: a user matrix $\boldsymbol{Q}$ and an item matrix $\boldsymbol{P}$, in which columns correspond to users and items, respectively, and rows to latent factors. Figure 1b shows an example with $r = 2$ latent factors, which in this case roughly correspond to action and romance. The predicted preference of user i for item j is given by the (i, j) entry of matrix product $\boldsymbol{Q}^T\boldsymbol{P}$ or, equivalently, by the inner product $\boldsymbol{q}^T\boldsymbol{p}$, where $\boldsymbol{q}$ denotes the i-th column of $\boldsymbol{Q}$ and $\boldsymbol{p}$ the j-th column of $\boldsymbol{P}$. The goal of a recommender system is to recommend to each user the items with a high predicted rating (among other criteria); we thus need to determine which entries are large. In the example of Figure 1b, we marked in bold face the top-3 items for each user.[3] In our terminology, user vectors correspond to queries and item vectors to probe vectors. We are interested in finding for each user the top-k items vectors with the largest inner products; we refer to this problem as Top-k-MIPS.

Another prominent application of matrix factorization models is in the area of open information extraction, which extracts and reasons about statements made in natural language text and other sources. Riedel et al. [2013], for example, construct a *fact matrix*, in which columns correspond to verbal phrases or relations (e.g., "was born in") and rows to (subject, object)-pairs (e.g., ("Einstein", "Ulm")). A nonzero entry indicates that the corresponding fact—a verbal phrase or relation with its subject and object—was observed in the available data. Matrix factorization techniques can be used to predict additional facts, spot unlikely facts, and reason about verbal phrases. As in recommender systems, these methods create factor matrices using a suitable model and subsequently determine the large entries in their product; here large entries cor-

[3]In practice, one may ignore items already seen or bought by a user.

respond to facts with a high predicted confidence. We refer to the problem of retrieving all entries above a specified threshold θ as Above-θ-MIPS.

In this article, we focus solely on the MIPS problem and are oblivious to how the input matrices have been created. In some applications, such as the ones above, MIPS is applied to the factor matrices obtained from some matrix factorization algorithm. Fast and scalable matrix factorization algorithms have been extensively studied in the literature [Makari et al. 2015; Recht and Ré 2011; Niu et al. 2011; Teflioudi et al. 2012] and the factorization itself is usually not a bottleneck (see Sec. 8.1 for some examples). Other applications, such as the ones mentioned in the introduction, do not make use of a prior matrix factorization step.

2.3. Problem Statement

Exact MIPS. We study two variants of exact MIPS. The first one searches for each vector $\boldsymbol{q} \in \boldsymbol{Q}$, the set of k vectors from $\boldsymbol{P}$ with the largest inner product with $\boldsymbol{q}$. Here k is application-defined. As discussed previously, this problem arises in recommender systems, where we want to retrieve the most relevant items (vectors of $\boldsymbol{P}$) for each user (vector of $\boldsymbol{Q}$).

Definition 2.1 (*Top-k-MIPS*). Given an integer $k > 0$, determine

$$\{ (\boldsymbol{q}, T_k(\boldsymbol{q})) \mid \boldsymbol{q} \in \boldsymbol{Q} \},$$

i.e., the top-k list for every query.

MIPS is often defined in the literature w.r.t. a single query, i.e., $\boldsymbol{Q} = (\boldsymbol{q})$. The Top-$k$-MIPS problem is then equivalent to top-k scoring with a linear scoring function $f(\boldsymbol{p}) = \boldsymbol{q}^T\boldsymbol{p}$ [Fagin et al. 2001]. In this article, we focus on the more general case in which $\boldsymbol{Q}$ has multiple columns (e.g., when queries arrive in batches). Top-k-MIPS is then equivalent to multi-query top-k scoring. The methods presented in this article can also be used in a single-query setting, of course. Finally, note that by reversing the roles of $\boldsymbol{Q}$ and $\boldsymbol{P}$, we can also find the top-k queries for each probe vector.

The second problem, termed Above-θ-MIPS, asks to retrieve all pairs of vectors with inner product above some application-defined threshold θ. This problem is useful, for example, to determine all high-confidence facts in an open relation extraction scenario.

Definition 2.2 (*Above-θ-MIPS*). Given a threshold $\theta > 0$, determine the set

$$\left\{ (\boldsymbol{q}, \boldsymbol{p}) \in \boldsymbol{Q} \times \boldsymbol{P} \mid \boldsymbol{q}^T\boldsymbol{p} \geq \theta \right\}$$

of large entries in $\boldsymbol{Q}^T\boldsymbol{P}$.

A simple solution to the above problems is to first compute $\boldsymbol{Q}^T\boldsymbol{P}$ and then select the entries above the threshold (for Above-θ-MIPS) or the k largest entries per row (for Top-k-MIPS). We refer to this approach as *Naive*; it has time complexity $O(mnr)$ and is infeasible for large problem instances. Recently, a number of algorithms for exact MIPS have been proposed [Ram and Gray 2012; Curtin et al. 2013; Curtin and Ram 2014]; all of these methods are based on suitable tree-based indexes built on $\boldsymbol{P}$ (see Sec. 7).

Approximate MIPS. Exact MIPS methods usually offer only limited speedup compared to naive search. Consequently, there has been a significant interest in designing methods for approximate MIPS [Shrivastava and Li 2014a; Bachrach et al. 2014; Shrivastava and Li 2014b; Neyshabur and Srebro 2014]. Such methods trade off quality of results and speed. In many applications, high-quality approximate results are acceptable. In recommender systems, for example, finding good recommendations fast may be preferable to finding the best recommendations slowly.

Consider an approximate Top-k-MIPS algorithm and fix a query $\boldsymbol{q}$. Denote by $\hat{T}_k$ the *approximate top-k list* being produced. We require $\hat{T}_k \subseteq \boldsymbol{P}$ and $|\hat{T}_k| = k$, i.e., approximate algorithms output k probe vectors (per query). For $1 \leq i \leq k$, denote by $\hat{s}_{(i)}$ the *approximate top-i score*, i.e., the i-th largest score in $\hat{T}_k$. Clearly, we must have $\hat{s}_{(i)} \leq s_{(i)}$; approximate results cannot be better than the exact results.

There are multiple ways to define the quality of the results of an approximate MIPS algorithm with respect to a query $\boldsymbol{q}$. Two commonly used metrics are *precision* (fraction of true results in output) and *recall* (fraction of all true results being output). Note that for Top-k-MIPS, both approximate and exact methods produce exactly k results, so that recall and precision coincide.

One disadvantage of using precision or recall for Top-k-MIPS is that they do not give any indication about the quality of the remaining ("false") vectors in the approximate top-k list. To see why this might be of importance, consider again the recommender system scenario. Generally, we prefer methods that give good "false" results over methods that give bad "false" results, and recall does not allow to distinguish these two cases. A measure that captures the difference between the result of the exact and the approximate method in absolute terms is the root mean square error (RMSE, [Bachrach et al. 2014]), defined as:

$$RMSE = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(s_{(i)} - \hat{s}_{(i)})^2}.$$

Alternatively, when all scores are positive ($s_{(k)} > 0$), we can quantify the difference relatively using the average relative error (ARE):

$$ARE = \frac{1}{k}\sum_{i=1}^{k}\frac{s_{(i)} - \hat{s}_{(i)}}{s_{(i)}}.$$

Since $\hat{s}_{(i)} \leq s_{(i)}$ for all i, an approximate method performs the better the larger the $\hat{s}_{(i)}$'s.

We define the recall/RMSE/ARE for a set of queries by taking the average of the recall/RMSE/ARE over all queries (i.e., the macro-average).

An approximate MIPS method that provides approximation guarantees takes as input an error bound on either recall, RMSE, or ARE and produces an approximate result that satisfies the specified bound (always, with high probability, or in expectation). Unfortunately, many of the existing approximate methods do not provide such guarantees and proceed in a best-effort manner instead. In Sec. 5, we propose a number of novel approximate methods that do provide error guarantees.

2.4. MIPS, Nearest Neighbor Search, and Cosine Similarity Search

The MIPS problem is closely related to the well-studied problems of nearest neighbor search in Euclidean space (NNS) and cosine similarity search (CSS). In this section, we discuss this relationship briefly; see Ram and Gray [2012] for more details.

The NNS problem is to find for a given, fixed query vector $\boldsymbol{q}$ a probe vector $\boldsymbol{p}^* \in \boldsymbol{P}$ such that

$$\boldsymbol{p}^* = \underset{\boldsymbol{p}\in\boldsymbol{P}}{\operatorname{argmin}} \|\boldsymbol{q} - \boldsymbol{p}\|^2 = \underset{\boldsymbol{p}\in\boldsymbol{P}}{\operatorname{argmax}} \left(\boldsymbol{q}^T\boldsymbol{p} - \frac{\|\boldsymbol{p}\|^2}{2}\right). \tag{1}$$

The CSS problem is to find a probe vector $\boldsymbol{p}^* \in \boldsymbol{P}$ such that

$$\boldsymbol{p}^* = \underset{\boldsymbol{p}\in\boldsymbol{P}}{\operatorname{argmax}} \frac{\boldsymbol{q}^T\boldsymbol{p}}{\|\boldsymbol{q}\|\|\boldsymbol{p}\|} = \underset{\boldsymbol{p}\in\boldsymbol{P}}{\operatorname{argmax}} \frac{\boldsymbol{q}^T\boldsymbol{p}}{\|\boldsymbol{p}\|}. \tag{2}$$

We distinguish two settings based on the L2 norms of the probe vectors:

(1) All vectors in $\boldsymbol{P}$ have equal L2 norm. In this case, the $\|\boldsymbol{p}\|$ part in Eqs. (1) and (2) is constant and does not affect the result. The MIPS problem is then equivalent to both NNS and CSS.
(2) The vectors in $\boldsymbol{P}$ have different L2 norms. MIPS is then different from both NNS and CSS.

In what follows, we focus on the case of vectors of unequal norm, i.e., the case where MIPS is different from NNS and CSS. One major challenge in this setting is that inner products do not adhere to the triangle inequality, which makes similarity search not directly applicable.

Two general directions have been proposed to solve the MIPS problem. *Customized methods* [Ram and Gray 2012; Curtin et al. 2013; Curtin and Ram 2014] exploit the structure of MIPS and develop suitable indexing techniques. It has been shown, however, that we can reformulate the MIPS problem as an NNS or CSS problem in a slightly higher dimensional space [Shrivastava and Li 2014a; Bachrach et al. 2014; Shrivastava and Li 2014b; Neyshabur and Srebro 2014]; see Sec. 7 for details. *Transformation-based methods* perform such a reformulation and subsequently apply an existing NNS or CSS method, respectively. In Sec. 8, we provide experimental insight into the relative performance of customized methods (including LEMP) and transformation-based methods.

3. The LEMP Framework

In this section, we outline the LEMP framework for exact and approximate MIPS. For presentation purposes, we first focus on the Above-θ-MIPS problem and turn to the Top-k-MIPS problem in Sec. 4.5.

3.1. L2 Norm and Direction

LEMP makes use of the decomposition of an inner product of two vectors $\boldsymbol{q}$ and $\boldsymbol{p}$ into an L2 norm and a direction part. Let $\boldsymbol{0} \neq \boldsymbol{v} \in \mathbb{R}^r$ and denote by $\bar{\boldsymbol{v}} = \boldsymbol{v}/\|\boldsymbol{v}\|$ the *normalization* of $\boldsymbol{v}$, i.e., the unit vector pointing in the direction of $\boldsymbol{v}$. Then

$$\boldsymbol{q}^T\boldsymbol{p} = \|\boldsymbol{q}\|\,\|\boldsymbol{p}\|\cos(\boldsymbol{q},\boldsymbol{p}), \tag{3}$$

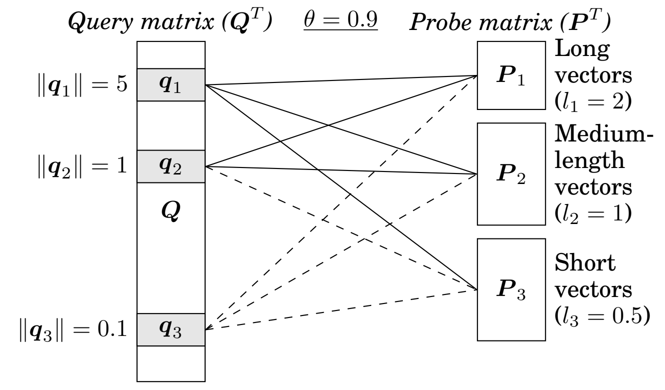
where $\cos(\boldsymbol{q},\boldsymbol{p}) = \bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \in [-1,1]$ denotes the cosine similarity between $\boldsymbol{q}$ and $\boldsymbol{p}$. As mentioned previously, the inner product coincides with the cosine similarity if $\boldsymbol{q}$ and $\boldsymbol{p}$ have unit norm. The problem of cosine similarity search is thus a special case of the MIPS problem.

By rewriting Eq. (3), we obtain for $\theta \in \mathbb{R}$

$$\boldsymbol{q}^T\boldsymbol{p} \geq \theta \iff \cos(\boldsymbol{q},\boldsymbol{p}) \geq \frac{\theta}{\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|}. \tag{4}$$

The inner product thus exceeds threshold θ if and only if the cosine similarity exceeds the modified threshold $\frac{\theta}{\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|}$, which depends on the L2 norms of $\boldsymbol{q}$ and $\boldsymbol{p}$. Our goal is to find pairs $(\boldsymbol{q},\boldsymbol{p}) \in \boldsymbol{Q} \times \boldsymbol{P}$ such that $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$. From Eq. 4, we conclude that:

(1) If $\boldsymbol{q}$ and $\boldsymbol{p}$ are short in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| < \theta$, we cannot have $\boldsymbol{q}^T\boldsymbol{p} > \theta$ since $\cos(\boldsymbol{q},\boldsymbol{p}) \in [-1,1]$ and $\theta/(\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|) > 1$. Such pairs do not need to be considered.
(2) If $\boldsymbol{q}$ and $\boldsymbol{p}$ are of intermediate L2 norm in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| \approx \theta$, then $\boldsymbol{q}^T\boldsymbol{p} > \theta$ if the cosine similarity $\cos(\boldsymbol{q},\boldsymbol{p})$ is large. Such pairs are best found using a cosine similarity search algorithm.
(3) If $\boldsymbol{q}$ and $\boldsymbol{p}$ are long in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| \gg \theta$, then $\boldsymbol{q}^T\boldsymbol{p} > \theta$ if their cosine similarity is not too small. Such pairs are best found using naive search.



$\theta_b(q)$	P_1	P_2	P_3
q_1	0.09	0.18	0.36
q_2	0.45	0.90	-
q_3	-	-	-

Local thresholds

Fig. 2: Illustration of LEMP's bucketization

Our discussion so far indicates that probe vectors with different L2 norms are best treated in different ways. LEMP exploits this observation as follows. It first groups the vectors of the probe matrix P into a set of small buckets, each consisting of vectors of roughly similar L2 norm, and then solves a cosine similarity search problem for each bucket. In particular, we ignore buckets with short vectors, use a suitable cosine similarity search algorithm for buckets with vectors of intermediate L2 norms, and use (a variant of) naive retrieval for buckets with long vectors. This allows us to prune large parts of the search space and handle the remaining part efficiently. In this article, we explore how far we can push this basic idea.

In more detail, denote by P_1, P_2, ..., P_s a set of s buckets and assume that the vectors in each bucket have roughly similar (but not necessarily equal) L2 norms. For each bucket P_b, $1 \leq b \leq s$, denote by $l_b = \max_{p \in P_b} \|p\|$ the L2 norm of its longest vector. Under our assumption, $l_b \approx \|p\|$ for all $p \in P_b$. Figure 2 shows a small example in which P has been divided into three buckets: P_1 holds long vectors (approximate and maximum L2 norm 2), P_2 medium-length vectors (1), and P_3 short vectors (0.5). Although there are only three buckets in this example, LEMP uses a large number of buckets in practice.

Fix some bucket P_b. From Eq. (4), we obtain that a necessary condition for $q^T p \geq \theta$ for $p \in P_b$ is that

$$\cos(q, p) = \bar{q}^T \bar{p} \geq \theta_b(q) \overset{\text{def}}{=} \frac{\theta}{\|q\| \, l_b}. \tag{5}$$

We refer to $\theta_b(q)$ as the *local threshold* of query q for bucket P_b. Our goal is thus to find all vectors $p \in P_b$ with a cosine similarity to q of at least $\theta_b(q)$. The local threshold allows us to determine how to best process bucket P_b, analogous to the discussion above. If $\theta_b(q) > 1$, we can prune the entire bucket since none of its vectors can potentially pass the threshold. If $\theta_b(q) \approx 1$, we use a suitable cosine similarity search algorithm for the bucket. Finally, if $\theta_b \ll 1$, we use naive retrieval.

Consider again the example of Figure 2 and assume a global threshold of $\theta = 0.9$. The figure highlights three query vectors q_1, q_2, and q_3 of decreasing L2 norms and gives the values of all local thresholds (or "-" if above 1, also indicated by dashed lines). For q_1, which is very long, all local thresholds are small so that naive retrieval is well suited for all buckets. For q_2, which is shorter, the local threshold is small for bucket P_1 (long vectors), large for bucket P_2 (medium-length vectors), and above 1 for bucket P_3 (short vectors). We use naive retrieval for P_1 and a suitable cosine similarity search

Algorithm 1 LEMP for the exact Above-θ-MIPS problem

```
Require: $\boldsymbol{Q}, \boldsymbol{P}, \theta$
Ensure: $S = \{ (\boldsymbol{q}, \boldsymbol{p}) \in \boldsymbol{Q} \times \boldsymbol{P} \mid \boldsymbol{q}^T \boldsymbol{p} \geq \theta \}$
1:  // Preprocessing phase
2:  Partition $\boldsymbol{P}$ into buckets $\boldsymbol{P}_1, \ldots, \boldsymbol{P}_s$ of similar L2 norms
3:  for all $b \in 1, 2, \ldots, s$ do                                  // for each bucket
4:      Sort, normalize, and index $\boldsymbol{P}_b$
5:      $l_b \leftarrow \max_{\boldsymbol{p} \in \boldsymbol{P}_b} \|\boldsymbol{p}\|$
6:  end for
7:
8:  // Search phase
9:  $S \leftarrow \emptyset$
10: for all $b \in 1, 2, \ldots, s$ do                                  // for each bucket
11:     for all $\boldsymbol{q} \in \boldsymbol{Q}$ do                                   // for each query
12:         $\theta_b(\boldsymbol{q}) \leftarrow \theta \,/\, (\|\boldsymbol{q}\|\, l_b)$                              // local threshold
13:         if $\theta_b(\boldsymbol{q}) \leq 1$ then                                        // prune?
14:             Pick a suitable retrieval alg. $A$ based on $\theta_b(\boldsymbol{q})$
15:             Use $A$ to obtain a set of candidates $C_b \supseteq \{ \boldsymbol{p} \in \boldsymbol{P}_b \mid \bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q}) \}$
16:             $S \leftarrow S \cup \{ (\boldsymbol{q}, \boldsymbol{p}) \mid \boldsymbol{p} \in C_b$ and $\boldsymbol{q}^T \boldsymbol{p} \geq \theta \}$          // verify candidates
17:         end if
18:     end for
19: end for
```

algorithm for $\boldsymbol{P}_2$. Bucket $\boldsymbol{P}_3$ is pruned. Finally, for $\boldsymbol{q}_3$, which is very short, all local thresholds exceed 1 so that all buckets are pruned.

3.2. Algorithm Description

Alg. 1 summarizes LEMP for exact Above-θ-MIPS. The algorithm consists of a *preprocessing phase* (lines 1–6) and a *search phase* (lines 8–19).

The preprocessing phase groups the columns of $\boldsymbol{P}$ into buckets of similar L2 norm (line 2). There are a number of ways to do this, but we chose a simple greedy strategy in our implementation. In particular, we first sort the columns of $\boldsymbol{P}$ by decreasing L2 norm,[4] scan the columns in order, and start a new bucket whenever the L2 norm of the current column falls below some threshold (e.g., 90% of l_b). We also make sure that buckets are neither too small nor too large. First, small buckets reduce the efficiency of LEMP due to bucket processing overheads; we thus ensure that buckets contain at least a certain number of vectors (30 in our implementation). The bucket processing overhead of large buckets is negligible. However, when buckets grow larger than the cache size, processing time is negatively affected. For this reason, we select a maximum bucket size that ensures that all relevant data structures fit into the processor cache.

After bucket boundaries have been obtained, we represent each vector $\boldsymbol{p}$ by two separate components: its L2 norm $\|\boldsymbol{p}\|$ and its direction $\bar{\boldsymbol{p}}$. We also store the vectors' column number in the original matrix (denoted `id`) and in the bucket (denoted `lid` for "local id"); see Figure 4a for an example. This layout allows us to access for each $\boldsymbol{p} \in \boldsymbol{P}_b$ both $\|\boldsymbol{p}\|$ and $\bar{\boldsymbol{p}}$ without further computation. We then create indexes on the contents of each bucket; we defer the discussion of indexing to Sec. 4. For our choice of indexes (Sec. 4.2 and 4.3), the overall preprocessing time, including index computation, is $O(rn \log n)$.

The search phase then iterates over buckets and query vectors. For each query, we compute the local threshold $\theta_b(\boldsymbol{q})$ (line 12) and prune buckets based on their L2 norm

[4]We also sort and normalize query vectors in a manner similar to the bucketization of $\boldsymbol{P}$.

(line 13). For each remaining bucket $\boldsymbol{P}_b$, we select a suitable retrieval algorithm (exact or approximate) based on the local threshold (line 14, cf. Sec. 4). The selected retrieval algorithm computes a set C_b of *candidate vectors*, potentially making use of the index data structures created during the preprocessing phase. We require for exact MIPS that the candidate set contains all vectors in $\boldsymbol{p} \in \boldsymbol{P}_b$ that pass the threshold ($\boldsymbol{q}^T\boldsymbol{p} \geq \theta$), but it may additionally contain a set of *spurious vectors* ($\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q})$ but $\boldsymbol{q}^T\boldsymbol{p} < \theta$). If LEMP is used for approximate MIPS, we lift this requirement, i.e., the candidate set may then miss some vectors that pass the threshold.[5] In both cases, a verification step (line 16) filters out spurious vectors by computing the actual values of the inner products $\boldsymbol{q}^T\boldsymbol{p}$ for all $\boldsymbol{p} \in C_b$.

The order of the two loops in the search phase of Alg. 1 is chosen to be cache-friendly. Since we process probe buckets in the outer loop and since probe buckets are small, their content remains in the cache for the entire inner loop. The inner loop itself scans query vectors sequentially; these vectors may not fit into the cache, but the sequential access pattern makes prefetching effective.

The power of LEMP to prune entire buckets in line 12 depends on the L2 norm distribution of the input vectors: generally, the more skewed the L2 norm distribution, the more probe buckets can be pruned. Even if bucket pruning is not particularly effective for a given problem instance, however, the organization of the probe vectors into buckets is still beneficial: it allows suitable cosine similarity search algorithms to be applied and is cache-friendly.

4. Exact MIPS

In this section, we propose and discuss a number of exact algorithms for the search phase of LEMP (line 15 of Alg. 1). Each algorithm takes as input a query vector $\boldsymbol{q} \in \boldsymbol{Q}$ and a bucket $\boldsymbol{P}_b$, and outputs a candidate set $C_b \subseteq \boldsymbol{P}_b$ using some pruning strategy. All algorithms first compute $\|\boldsymbol{q}\|$ and $\bar{\boldsymbol{q}}$; cf. Figure 4d.

We discuss two kinds of algorithms: those that make use of only the L2 norm information to prune candidate vectors and those that use the normalized vectors as well. For the first category, we propose the NORM algorithm (Sec. 4.1), which is a simple variant of the naive algorithm that takes L2 norm information into account. Existing cosine similarity search algorithms (e.g., [Bayardo et al. 2007]) as well as TA fall in the second category. We additionally propose two novel methods, which are specially tailored for vectors of medium dimensionality. The COORD algorithm (Sec. 4.2) applies coordinate-based pruning strategies. The ICOORD algorithm (Sec. 4.3) is based on COORD but uses a more effective (but also more expensive) incremental pruning strategy.

4.1. Norm-Based Pruning

Recall that the vectors in bucket $\boldsymbol{P}_b$ are sorted by decreasing L2 norm during preprocessing (see also Figure 4a). Further observe from Eq. (3) that whenever $\|\boldsymbol{q}\|\,\|\boldsymbol{p}\| < \theta$, so is $\boldsymbol{q}^T\boldsymbol{p}$. Putting both together, NORM scans the bucket $\boldsymbol{P}_b$ in order. When processing vector $\boldsymbol{p}$, we check whether $\|\boldsymbol{p}\| \geq \theta/\|\boldsymbol{q}\|$; we precompute $\theta/\|\boldsymbol{q}\|$ to make this check efficient. If $\boldsymbol{p}$ qualifies, we add it to the candidate set C_b. Otherwise, we stop processing bucket $\boldsymbol{P}_b$ and immediately output C_b.

Consider for example a bucket $\boldsymbol{P}_b$ as shown in Figure 4a, query vector $\boldsymbol{q} = (1,1,1,1)^T$, and threshold $\theta = 3.8$. We have $\|\boldsymbol{q}\| = 2$ and $\theta/\|\boldsymbol{q}\| = 1.9$ so that we obtain $C_b = \{\,1,2,3\,\}$. (Here and in the following, we give C_b in terms of local identifiers (lid) for improved readability.)

[5]We instead require the candidate set is "good enough" to satisfy user-specified bounds on recall, RMSE or ARE.

Since LEMP already organizes and prunes buckets by L2 norm, we do not expect NORM to be particularly effective. In fact, NORM degenerates to the naive algorithm in all but one bucket (the "last" bucket that has not been pruned). Nevertheless, since NORM has low overhead and a sequential access pattern, it is an effective method when buckets are small or the local threshold is low (i.e., when coordinate-based pruning is not effective).

4.2. Coordinate-Based Pruning

We now proceed to pruning strategies based on the direction (but not L2 norm) of the query vector. The key idea is to retain only those vectors from $\boldsymbol{P}_b$ in C_b that point in a similar direction as $\boldsymbol{q}$. In particular, we aim to find all $\boldsymbol{p} \in \boldsymbol{P}_b$ with high cosine similarity to $\boldsymbol{q}$, i.e.,

$$\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} = \cos(\boldsymbol{q}, \boldsymbol{p}) \geq \theta_b(\boldsymbol{q}). \tag{6}$$

Note the usage of normalized vectors here; L2 norm information is not taken into account.

Let $\bar{\boldsymbol{q}} = (\bar{q}_1, \ldots, \bar{q}_r)^T$ and $\bar{\boldsymbol{p}} = (\bar{p}_1, \ldots, \bar{p}_r)^T$. Note that $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}$ achieves its maximum value for $\bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}$ since then $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}^T \bar{\boldsymbol{q}} = \|\bar{\boldsymbol{q}}\|^2 = 1$. In other words, $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}$ is maximized when both vectors agree on all their coordinates. Based on this observation, the key idea of the COORD algorithm is to prune $\bar{\boldsymbol{p}}$ if one of its coordinates deviates too far from the respective coordinate in $\bar{\boldsymbol{q}}$. In more detail, we obtain for each coordinate $f \in [r]$ a lower bound $L_f(\bar{\boldsymbol{q}})$ and an upper bound $U_f(\bar{\boldsymbol{q}})$ on $\bar{p}_f$. If $L_f \leq \bar{p}_f \leq U_f$, we say that $\bar{p}_f$ is *feasible*; otherwise $\bar{p}_f$ is *infeasible*. The bounds are chosen such that whenever a coordinate f of $\boldsymbol{p}$ is infeasible, then $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} < \theta_b(\boldsymbol{q})$ so that $\boldsymbol{p}$ can be pruned from the candidate set.

In what follows, we provide lower and upper bounds, discuss their effectiveness, and propose the COORD algorithm that exploits them.

Bounding Coordinates. Pick some coordinate $f \in [r]$; we refer to f as a *focus coordinate*. Denote by $\bar{\boldsymbol{q}}_{\text{-}f} = \{\, \bar{q}_1, \ldots, \bar{q}_{f-1}, \bar{q}_{f+1}, \ldots, \bar{q}_r \,\}$ the vector obtained by removing coordinate f from $\bar{\boldsymbol{q}}$, similarly $\bar{\boldsymbol{p}}_{\text{-}f}$. Note that $\bar{\boldsymbol{q}}_{\text{-}f}$ and $\bar{\boldsymbol{p}}_{\text{-}f}$ generally have L2 norms less than 1. Now we rewrite Eq. (6) as follows

$$\begin{aligned}
\theta_b(\boldsymbol{q}) &\leq \bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} \\
&= \bar{q}_f \bar{p}_f + \bar{\boldsymbol{q}}_{\text{-}f}^T \bar{\boldsymbol{p}}_{\text{-}f} \\
&= \bar{q}_f \bar{p}_f + \|\bar{\boldsymbol{q}}_{\text{-}f}\| \, \|\bar{\boldsymbol{p}}_{\text{-}f}\| \cos(\bar{\boldsymbol{p}}_{\text{-}f}, \bar{\boldsymbol{q}}_{\text{-}f}) \\
&\leq \bar{q}_f \bar{p}_f + \|\bar{\boldsymbol{q}}_{\text{-}f}\| \, \|\bar{\boldsymbol{p}}_{\text{-}f}\| \\
&= \bar{q}_f \bar{p}_f + \sqrt{1 - \bar{q}_f^2} \sqrt{1 - \bar{p}_f^2},
\end{aligned} \tag{7}$$

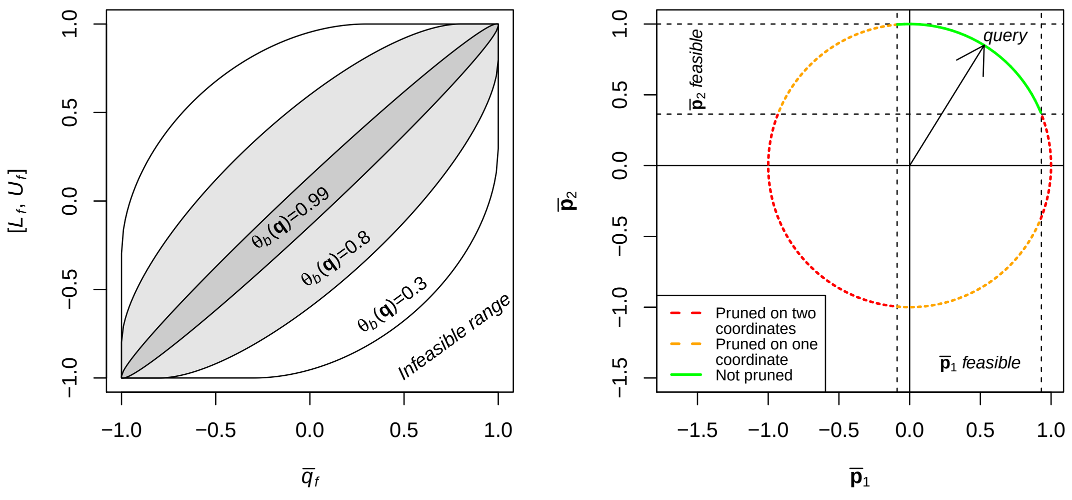
where we used Eq. (3), the fact that the cosine similarity cannot exceed 1, and the property $\|\bar{\boldsymbol{q}}\| = \|\bar{\boldsymbol{p}}\| = 1$.

We now solve the resulting inequality $\theta_b(\boldsymbol{q}) \leq \bar{q}_f \bar{p}_f + (1 - \bar{q}_f^2)^{1/2} (1 - \bar{p}_f^2)^{1/2}$ for $\bar{p}_f$ and obtain solutions

$$\bar{p}_f \in [L_f^A, U_f^A] = \begin{cases} [\theta_b(\boldsymbol{q})/\bar{q}_f, 1] & \bar{q}_f > 0 \\ [-1, \theta_b(\boldsymbol{q})/\bar{q}_f] & \bar{q}_f < 0 \end{cases} \tag{8}$$

$$\bar{p}_f \in (L_f^B, U_f^B) = \left(\bar{q}_f \theta_b(\boldsymbol{q}) - \sqrt{(1 - \theta_b(\boldsymbol{q})^2)(1 - \bar{q}_f^2)}, \bar{q}_f \theta_b(\boldsymbol{q}) + \sqrt{(1 - \theta_b(\boldsymbol{q})^2)(1 - \bar{q}_f^2)} \right) \tag{9}$$

$$\bar{p}_f \in [L_f^C, U_f^C] = [-1, 1], \text{ if } \bar{q}_f = 0, \theta_b(\boldsymbol{q}) \leq 0. \tag{10}$$



(a) Feasible regions of $\bar{\boldsymbol{p}}_f$ for various values of $\theta_b(\boldsymbol{q})$. (b) Bounds on example query for $\theta_b(\boldsymbol{q}) = 0.8$

Fig. 3: Usage of feasibility bounds.

Here Eq. (8) is only a valid solution to Eq. (7) when $L_f^A \le U_f^A$, i.e., we ignore it whenever $L_f^A > U_f^A$. In addition Eq. (10) provides no pruning power to our algorithm. We therefore directly skip query coordinates of zero value whenever $\theta_b(\boldsymbol{q}) \le 0$. The feasible region is thus given by:

$$L_f = \begin{cases} \min(L_f^A, L_f^B) & \text{if } L_f^A \le U_f^A \\ L_f^B & \text{otherwise} \end{cases} \tag{11}$$

$$U_f = \begin{cases} \max(U_f^A, U_f^B) & \text{if } L_f^A \le U_f^A \\ U_f^B & \text{otherwise} \end{cases} \tag{12}$$

Note that if the L2 norms of the vectors within a bucket vary strongly, we are forced to use a low local threshold $\theta_b(\boldsymbol{q})$, which in turn results in looser bounds. This undesirable behavior is avoided by LEMP since it constructs (small) buckets that contain vectors with similar L2 norms. The effectiveness of our bounds—and of using normalization and subsequent coordinate-based pruning in general—is thus particularly effective in the context of our LEMP framework.

Effectiveness of Bounds. To gain some insight into the effectiveness of LEMP's bounds, we plot the *feasible region* $[L_f, R_f]$ for various choices $\theta_b(\boldsymbol{q})$ in Figure 3a. The x-axis corresponds to the value of $\bar{q}_f$, the y-axis to the lower and upper bounds, and the various oval-shaped gray regions to feasible regions. Note that $-1 \le \bar{q}_f, \bar{p}_f \le 1$.

The pruning power of the bounds depends on both the value of $\theta_b(\boldsymbol{q})$ and on the properties of matrices $\boldsymbol{Q}$ and $\boldsymbol{P}$. First, the larger the local threshold $\theta_b(\boldsymbol{q})$, the smaller the feasible region and the more vectors can be pruned. In fact, for large values of $\theta_b(\boldsymbol{q})$, the feasible region is small across the entire value range of $\bar{q}_f$. Second, the size of the feasible region decreases as the absolute value of $\bar{q}_f$ increases. This decrease is more pronounced when the local threshold is small. Note that a small feasible region may or may not lead to effective pruning; the value distribution of $\boldsymbol{P}_b$ is also important. Nevertheless, the smaller the feasible region, the more effective the pruning can be.

Based on the observations above, we conclude that the bounds can effectively prune a vector $\bar{\boldsymbol{p}}$ with $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} < \theta_b(\boldsymbol{q})$ when $\theta_b(\boldsymbol{q})$ is large or when there is some coordinate f for which only one of $\bar{q}_f$ or $\bar{p}_f$ takes a large value. Since all vectors are L2-normalized, the latter property holds if $\bar{\boldsymbol{q}}$ or $\bar{\boldsymbol{p}}$ is sufficiently sparse or has a skewed value distribution. If neither holds and $\theta_b(\boldsymbol{q})$ is small, an algorithm such as NORM or ICOORD (see Sec. 4.3) may be a more suitable choice.

To make pruning more effective, we may consider multiple focus coordinates and require probe vectors to be feasible for all these coordinates. Fig. 3b illustrates this idea for $r = 2$ dimensions and $\theta_b(\boldsymbol{q}) = 0.8$. The normalized query vector is depicted in black. All normalized probe vectors in the bucket (potential candidates) lie on the unit sphere. The dotted vertical and horizontal lines mark the feasible regions for coordinates 1 and 2, respectively. The green (solid) part of the unit circle corresponds to probe vectors that are feasible for both focus coordinates. These vectors will not be pruned by COORD. The orange (dashed) part corresponds to probe vectors that are infeasible in one and only one of the two coordinates. These vectors are pruned by COORD when this coordinate is included in the set of focus coordinates. Finally, the red (dotted) part corresponds to probe vectors that are infeasible with respect to both coordinates; they will always be pruned by COORD. The figure indicates that we obtain the highest pruning power when we test for feasibility on both coordinates because only then both orange (dashed) regions are pruned. In the following, we describe in more detail how COORD makes use of this observation.

Exploiting Bounds. The COORD algorithm makes use of the feasible region derived in the previous section to prune unpromising candidates. To do so, LEMP creates indexes for each probe bucket $\boldsymbol{P}_b$ during its preprocessing phase. In the case of COORD, we create r sorted lists $I_1, \ldots, I_r$, one for each coordinate of the vectors in $\boldsymbol{P}_b$. Each entry in list I_f is a (lid, $\bar{p}_f$)-pair, where as before lid is a bucket-local identifier for the corresponding vector $\bar{\boldsymbol{p}}$. As in Fagin et al.'s threshold algorithm (TA, [Fagin et al. 2001]), from which our index is inspired, the lists are sorted in decreasing order of $\bar{p}_f$. Figure 4c shows the sorted-list index for the example bucket given in Figure 4a. Although index construction is generally light-weight and fast, LEMP constructs indexes lazily on first use to further reduce computational cost. Buckets with very short vectors, for example, may always be pruned and thus do not need to be indexed.

COORD is summarized as Alg. 2. It takes as input a bucket $\boldsymbol{P}_b$, a query $\boldsymbol{q}$, the global and local thresholds $(\theta, \theta_b(\boldsymbol{q}))$, the bucket indexes $I_1, \ldots, I_r$, and a set of focus coordinates $F \subseteq [r]$. We discuss the algorithm using the example of Figure 4 with $\theta = 0.9$. Consider the query $\boldsymbol{q}$ shown in Figure 4d as well as the corresponding inner products shown in Figure 4b. We have $\theta_b(\boldsymbol{q}) = 0.9/(0.5{\cdot}2) = 0.9$, coincidentally agreeing with the global threshold. Observe that vectors 1 and 5 pass the local threshold $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q})$, but only vector 1 additionally passes the global threshold $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$.

COORD does not compute and enforce the bounds for each coordinate, since this can be expensive, but uses a suitable subset $F \subseteq [r]$ of focus coordinates; see below. For each focus coordinate $f \in F$, COORD computes the feasible region $[L_f, U_f]$ (line 3) and determines the start and end of the corresponding *scan range* in sorted list I_f via binary search for U_f and L_f, respectively (line 4). Vectors outside the scan range violate the bound on coordinate f. In the example of Figure 4, we used $F = \{1, 4\}$. The bounds are shown in Figure 4d and the corresponding scan ranges in I_1 and I_4 are shown in bold face in Figure 4c.

lid	id	$\|p\|$	$\bar{p}$			
1	23	2.0	0.58	0.50	0.40	0.50
2	43	1.9	0.98	0	0	0.20
3	12	1.9	0.53	0	0	0.85
4	54	1.8	0.35	0.93	0	0.10
5	18	1.8	0.58	0.50	0.40	0.50
6	20	1.8	0.30	-0.40	0.81	-0.30

(a) Organization of bucket $\boldsymbol{P}_b$

$\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}$	$\boldsymbol{q}^T\boldsymbol{p}$
0.97	**0.97**
0.79	0.75
0.80	0.76
0.56	0.52
0.97	0.87
0.26	0.23

(b) Results for query $\boldsymbol{q}$ of (d)

I_1

lid	$\bar{p}_1$
2	0.98
1	**0.58**
5	**0.58**
3	**0.53**
4	**0.35**
6	0.30

I_2

lid	$\bar{p}_2$
4	0.93
1	0.50
5	0.50
2	0
3	0
6	-0.40

I_3

lid	$\bar{p}_3$
6	0.81
1	0.40
5	0.40
2	0
3	0
4	0

I_4

lid	$\bar{p}_4$
3	0.85
1	**0.50**
5	**0.50**
2	**0.20**
4	**0.10**
6	-0.30

(c) Sorted-list index (bold rows show scan range for $\boldsymbol{q}$)

$\|\boldsymbol{q}\|$	$\bar{\boldsymbol{q}}$			
0.5	0.70	0.3	0.4	0.51
$[L_f, U_f]$	$[0.32, 0.94]$	-	-	$[0.09, 0.83]$

(d) Query $\boldsymbol{q}$ and feasible region for focus coordinates

lid	c
1	**2**
2	1
3	1
4	**2**
5	**2**
6	0

$C_b = \{1, 4, 5\}$

(e) CP array

lid	c	$\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F$	$\|\boldsymbol{p}_F\|^2$	u	$\theta_{\boldsymbol{p}}(\boldsymbol{q})$
1	**2**	**0.66**	**0.59**	**0.32**	**0.9**
2	1	0.10	0.04	0.49	0.95
3	1	0.37	0.28	0.43	0.95
4	2	0.30	0.13	0.47	1
5	2	0.66	0.59	0.32	1
6	0	-	-	-	1

$C_b = \{1\}$

(f) Extended CP array

Fig. 4: Illustration of LEMP as well as the COORD and ICOORD retrieval algorithms for $\theta = 0.9$ and $F = \{1, 4\}$

COORD subsequently scans the scan range of each sorted list I_f, $f \in F$, in sequence (line 5) and maintains a *candidate-pruning array* (CP array, line 6). The CP array contains for each vector $\bar{\boldsymbol{p}} \in \boldsymbol{P}_b$ with local identifier lid a counter $c[lid]$ that indicates how often the vector has been seen so far. The CP array of our running example is shown in Figure 4e (with an additional lid column for improved readability). After completing all scans, COORD includes into C_b all those vectors $\bar{\boldsymbol{p}} \in \boldsymbol{P}_b$ that qualified on all focus coordinates, i.e., for which $c[lid] = |F|$ (line 9). In our example, $C_b = \{1, 4, 5\}$

Algorithm 2 The COORD algorithm

```
Require: $\boldsymbol{q}, \boldsymbol{P}_b, \theta, \theta_b(\boldsymbol{q}), F \subseteq [r], I_1, \ldots, I_r$
Ensure: $C_b \supseteq \{\, \boldsymbol{p}_j \in \boldsymbol{P}_b \mid \bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}_j \geq \theta_b(\boldsymbol{q}) \,\}$
1: $c \leftarrow$ empty CP array
2: for all $f \in F$ do
3:     Calculate feasible region $[L_f, U_f]$
4:     Determine corresponding scan range in sorted list $I_f$
5:     for all $lid$ in scan range of $I_f$ do
6:         $c[lid] \leftarrow c[lid] + 1$                    // maintain CP array
7:     end for
8: end for
9: $C_b = \{\, lid \mid c[lid] = |F| \,\}$                    // filter
```

since only those three vectors occurred in both scan ranges. In particular, vectors 2,3 and 6 are (correctly) excluded because they appear in only one or none scan range.

We now turn to the question of how to choose the focus set F. One option is to simply set $F = [r]$. However, processing sorted lists can get expensive if F is large or contains coordinates for which pruning is not effective, i.e., for which a large fraction of the corresponding sorted lists needs to be scanned. We make use of a focus-set size parameter ϕ, typically in the range of 1–5; we discuss the choice of ϕ in Sec. 4.4. COORD then uses the ϕ coordinates of $\bar{\boldsymbol{q}}$ with largest absolute value as focus coordinates. The reasoning behind this choice is that large coordinates will lead to the smallest feasible region (cf. Sec. 4.2); the hope is that they also lead to a small scan ranges and a small candidate set.

To summarize, COORD builds indexes only if needed and uses only a subset of the entries in a subset of the sorted-list indexes. The index scan itself is light-weight; it accesses solely the lid part of the lists and increases the counters of the CP array. Also note that the bounds we use for determining the scan range of the lists are simple and relatively cheap to compute. This is important since these bounds need to be computed per query, per bucket, and per focus coordinate. Implementation details also matter; see online Appendix A.

4.3. Incremental Coordinate-Based Pruning

COORD scans the sorted-list indexes to find the set of vectors that qualify in each coordinate $f \in F$, i.e., fall in region $[L_f, U_f]$. Other than checking feasibility, the actual values in the scanned lists are ignored. In contrast, the incremental pruning algorithm ICOORD makes use of the $\bar{p}_f$ values as well: It maintains information that allows it to prune additional vectors. Such an approach is generally more expensive than COORD, but the increase in pruning power may offset the costs.

When we derived the bounds of a coordinate f of COORD, we assumed that $\cos(\bar{\boldsymbol{q}}_{\text{-}f}, \bar{\boldsymbol{p}}_{\text{-}f}) = 1$. This is a worst-case assumption; in general, $\cos(\bar{\boldsymbol{q}}_{\text{-}f}, \bar{\boldsymbol{p}}_{\text{-}f})$ will be less (and often much less) than 1. Intuitively, a vector $\bar{\boldsymbol{p}}$ that qualifies barely in all coordinates often does not constitute an actual result. Recall our ongoing example of Figure 4. Here vector 4 barely qualifies in both indexes I_1 and I_4 and is thus included into the candidate set of COORD. Vector 4 does not pass the local threshold, however, since $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}}_4 = 0.56 < 0.9$. COORD is blind to this behavior.

Another potential drawback of COORD is that it does not (and cannot) take into consideration the L2 norm distribution of the vectors in each bucket. In the example of Figure 4, normalized vectors 1 and 5 are identical and both pass the local threshold.

However, since vector 1 is has slightly larger L2 norm than vector 5, only vector 1 passes the global threshold and thus the verification step of LEMP .

Similar to COORD, ICOORD scans the scan ranges of the sorted lists of the focus coordinates. To address the above issues, however, ICOORD additionally maintains a partial inner product for each of the vectors that it encounters. Generalizing our previous notation, denote by $\bar{\boldsymbol{q}}_F$ ($\bar{\boldsymbol{q}}_{\text{-}F}$) the values of the focus coordinates (of all other coordinates) of the query vector; similarly, $\bar{\boldsymbol{p}}_F$ and $\bar{\boldsymbol{p}}_{\text{-}F}$. We obtain

$$\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + \bar{\boldsymbol{q}}_{\text{-}F}^T\bar{\boldsymbol{p}}_{\text{-}F} \leq \bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + \|\bar{\boldsymbol{q}}_{\text{-}F}\|\,\|\bar{\boldsymbol{p}}_{\text{-}F}\|.$$

Since vectors are normalized, the right-hand side can be computed from $\bar{\boldsymbol{q}}_F$ and $\bar{\boldsymbol{p}}_F$ only. Denote the resulting upper bound on the "unseen" part $\bar{\boldsymbol{q}}_{\text{-}F}^T\bar{\boldsymbol{p}}_{\text{-}F}$ of the inner product $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}$ by

$$u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F) = \|\bar{\boldsymbol{q}}_{\text{-}F}\|\,\|\bar{\boldsymbol{p}}_{\text{-}F}\| = \sqrt{1-\|\bar{\boldsymbol{q}}_F\|^2}\sqrt{1-\|\bar{\boldsymbol{p}}_F\|^2}.$$

Then $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \leq \bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F)$. In order to compute this bound, ICOORD uses an *extended CP array*, which maintains for each probe vector in addition to the frequency counters of COORD (line 6 of Alg. 2) the quantities $\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F$ and $\|\bar{\boldsymbol{p}}_F\|^2 = \sum_{f\in F}\bar{p}_f^2$. After the extended CP array has been computed, ICOORD includes into the candidate set only those vectors $\bar{\boldsymbol{p}}$ that satisfy

$$\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + u(\|\bar{\boldsymbol{q}}_F\|, \|\bar{\boldsymbol{p}}_F\|) \geq \theta_{\boldsymbol{p}}(\boldsymbol{q}) \overset{\text{def}}{=} \frac{\theta}{\|\boldsymbol{p}\|\|\boldsymbol{q}\|}. \tag{13}$$

Here $\theta_{\boldsymbol{p}}(\boldsymbol{q})$ is an improved, probe vector-specific local threshold; it holds $\theta_{\boldsymbol{p}}(\boldsymbol{q}) \geq \theta_b(\boldsymbol{q})$. This improved local threshold cannot be used by the COORD algorithm.

Figure 4f shows the extended CP array for our running example (to the left of the double vertical lines) as well as the quantities involved in the above pruning condition (to the right; here we write u for $u(\|\bar{\boldsymbol{q}}_F\|, \|\bar{\boldsymbol{p}}_F\|)$). For example, for vector 1, $\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F = 0.58\cdot 0.70 + 0.50\cdot 0.51 = 0.66$ and $u = \sqrt{1-(0.58^2+0.50^2)}\cdot\|\bar{\boldsymbol{q}}_{\text{-}F}\|$. The quantity $\|\bar{\boldsymbol{q}}_{\text{-}F}\|$ (not shown in Figure 4f) is independent of the probe vectors and thus only computed once. In our example, $\|\bar{\boldsymbol{q}}_{\text{-}F}\| = \sqrt{1-(0.70^2+0.51^2)} = 0.5$. As can be seen in the example, filter condition $\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + u \geq \theta_{\boldsymbol{p}}(\boldsymbol{q})$ is passed only by vector 1; thus $C_b = \{\,1\,\}$. Note that the rows of vector 5 and vector 1 agree in the extended CP array; our improved local threshold (0.9 for vector 1 vs. 1 for vector 5), however, allows us to correctly prune vector 5 but retain vector 1.

4.4. Algorithm Selection

Before processing a bucket $\boldsymbol{P}_b$, LEMP needs to decide which retrieval algorithm to use. We have already given some guidance for this choice above: Norm-based pruning is suitable for buckets with a skewed L2 norm distribution, whereas coordinate-based pruning is suitable for large local thresholds and/or data with a skewed value distribution. In general, the choice of a suitable algorithm is data-dependent.

LEMP uses a simple, pragmatic method for algorithm selection: it samples a small set of query vectors and tests the different methods for each bucket. We observe the wall-clock times obtained by the various methods and select a threshold t_b for each bucket: whenever $\theta_b(\boldsymbol{q}) < t_b$, LEMP will use NORM, otherwise it uses coordinate-based pruning. For setting t_b, we simply pick the value that minimizes the runtime on the sampled query vectors.

We proceed similarly to select for each bucket b the parameter ϕ_b for the number of focus coordinates in coordinate-based pruning. We generally explore ϕ_b values in increasing order, i.e., starting with 1 and ending with some upper bound (e.g., 10). To speed up this process, we employ two heuristics. First, we stop exploring larger values for ϕ_b if the performance with the currently tested ϕ value is more than 10% worse than the best performance so far. Second, we use the best ϕ-value of bucket b as a starting point for tuning bucket $b + 1$. We then start exploring the performance for values of ϕ left and right to this initial value, using the same stopping criterion as above.

The cost of this sample-based profiling step is negligible since the number of query vectors is large; the overall running time is dominated by the time required to process $\boldsymbol{Q}$ in its entirety.

More elaborate approaches for algorithm selection are possible, e.g., some form of reinforcement learning. Our experiments suggest, however, that even the simple selection criterion outlined above gives promising results.

4.5. Solving the Top-k-MIPS Problem

Our discussion so far has focused on the Above-θ-MIPS problem; we now proceed to the discussion of the Top-k-MIPS problem. Recall that given a query vector $\boldsymbol{q}$, the Top-k-MIPS problem asks for the vectors $\boldsymbol{p} \in \boldsymbol{P}$ that attain the k largest inner products $\boldsymbol{q}^T\boldsymbol{p}$. Top-$k$-MIPS can be used in recommender systems, for example, to retrieve the best k item recommendations for each user.

The Top-k-MIPS problem is related to the Above-θ-MIPS problem as follows. Fix a query vector $\boldsymbol{q}$ and denote by $s_{(k)}$ its top-k score. The solution of the Top-k-MIPS problem coincides with the solution of the Above-θ^*-MIPS algorithm with threshold $\theta^* = s_{(k)}$. We do not know $s_{(k)}$ and thus θ^*, however, and instead make use of a running lower bound $\hat{\theta} \leq \theta^*$. The value of $\hat{\theta}$ increases as the algorithm proceeds.

In more detail, we take the k longest vectors of $\boldsymbol{P}$ (all located at the beginning of bucket $\boldsymbol{P}_1$) and compute their inner product with $\boldsymbol{q}$. The smallest so-obtained value is our initial choice of $\hat{\theta}$. We then run the Above-θ-MIPS algorithm with threshold $\hat{\theta}$ on the first bucket, determine the top-k answers in the result, and update $\hat{\theta}$ accordingly. In more detail, we set $\hat{\theta}$ to the value of the k-th largest inner product found so far. This process is iterated over the subsequent buckets until $\hat{\theta}$ becomes so large that LEMP prunes the next bucket. At this point, we output the current top-k vectors as a result. This strategy is effective because (1) LEMP organizes buckets by decreasing L2 norm so that we expect the top-k values to appear in the top-most buckets, and (2) bucket sizes are small (cache-resident) so that the threshold $\hat{\theta}$ is increased frequently. The above algorithm is guaranteed to produce the correct result because $\hat{\theta} \leq \theta^* = s_{(k)}$ by construction. If a bucket contains a vector $\boldsymbol{p}$ with $\boldsymbol{q}^T\boldsymbol{p} \geq \theta^*$, then $\boldsymbol{q}^T\boldsymbol{p} \geq \hat{\theta}$ and we are guaranteed to add $\boldsymbol{p}$ to the candidate set (and retain it).

Note that the L2 norm of $\boldsymbol{q}$ does not affect the result of the Top-k-MIPS problem. We thus simplify the bounds used by the algorithms by normalizing $\boldsymbol{q}$ upfront.

5. Approximate MIPS

We now turn attention to approximate MIPS with LEMP, which allows us to realize further performance gains. We propose four different algorithms. The LEMP-LSHA algorithm (Sec. 5.2) is based on locality-sensitive hashing and offers a probabilistic guarantee on recall for both Above-θ-MIPS and Top-k-MIPS. The LEMP-ABS and LEMP-REL algorithms (Sec. 5.3) offer (non-probabilistic) bounds on the RMSE and ARE, respectively, and can be used with arbitrary exact bucket algorithms. Finally, the LEMP-HYB algorithm combines these approaches for further performance gains, and provides a weaker form of quality guarantee.

5.1. Preliminaries: Locality-Sensitive Hashing for Cosine Similarity Search

We start with a brief, high-level review of locality-sensitive hashing (LSH, [Gionis et al. 1999]) for approximate cosine similarity search. LSH is a popular and highly efficient technique for this problem; we are thus interested in adapting it to process each of LEMP's buckets (Sec. 3).

The key idea of LSH is to use a random hash function to assign probe vectors to bins. The hash function is chosen such that vectors with high cosine similarity are more likely to end up in the same bin than vectors with low cosine similarity. LSH is used as follows: we first group probe vectors into bins according to their hash values in a preprocessing phase. For each query vector $\boldsymbol{q}$, we determine $\boldsymbol{q}$'s bin using the same hash function, and consider as candidates each of the probe vectors in the corresponding bin; all other bins are ignored. Since vectors with high cosine similarity are (more) likely to end up in the same bin, the so-retrieved candidate set is biased towards vectors that are similar to $\boldsymbol{q}$.

In more detail, we use hash functions based on random hyperplanes [Charikar 2002], which work as follows. We first independently obtain r samples from the standard Normal distribution to form an r-dimensional vector $\boldsymbol{u}$. We view $\boldsymbol{u}$ as the normal vector of a random hyperplane. We then assign each probe vector to a bucket depending on which "side" of the hyperplane it lies, i.e.,

$$h_{\boldsymbol{u}}(\boldsymbol{p}) = \begin{cases} 1 & \boldsymbol{u}^T\boldsymbol{p} \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

One can show that for all pairs of vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^r$ [Charikar 2002]:

$$Pr_{\boldsymbol{u}}[h_{\boldsymbol{u}}(\boldsymbol{x}) = h_{\boldsymbol{u}}(\boldsymbol{y})] = 1 - \frac{\arccos(\bar{\boldsymbol{x}}^T\bar{\boldsymbol{y}})}{\pi} \tag{14}$$

Note that the right-hand side $s(\boldsymbol{x}, \boldsymbol{y}) = 1 - \arccos(\bar{\boldsymbol{x}}^T\bar{\boldsymbol{y}})/\pi$ is not identical to the cosine similarity $\bar{\boldsymbol{x}}^T\bar{\boldsymbol{y}}$. It is, however, a monotonically increasing function of the cosine similarity, which is sufficient for cosine similarity search.

To make LSH effective, we use l independent hash functions, where $l > 0$ is a parameter. For each vector $\boldsymbol{p}$, we concatenate its hash values into an l-bit binary code, called *signature*. Each of the 2^l potential signatures corresponds to a bin. The parameter l controls both cost and recall: If l is increased, more hash values need to be computed (increasing computational cost), fewer vectors are stored in each bin in expectation (reducing computational costs), and finally two similar vectors are less likely to be mapped to the same bin (reducing recall). To combat the loss in recall, the entire process can be repeated $\mathcal{L}$ times, where $\mathcal{L} > 0$ is another parameter. We then process each query on each of the $\mathcal{L}$ repetitions and union the results. Note that LSH is only effective when the number of probe vectors is larger than $l\mathcal{L}$ (because otherwise we need to compute more inner products to obtain hash values than naive search needs to obtain the exact result).

The most effective combination of l and $\mathcal{L}$ is generally data-dependent. If l is fixed, however, we can determine a suitable value for $\mathcal{L}$ according to the following theorem.

THEOREM 5.1 ([XIAO ET AL. 2011; SATULURI AND PARTHASARATHY 2012]). *Let $l > 0$, $\theta > 0$ and $0 < R < 1$. Consider an LSH data structure constructed on $\boldsymbol{P}$ using signatures of length l and*

$$\mathcal{L}(\theta) = \left\lceil \frac{\log(1-R)}{\log(1-[1-\arccos(\theta)/\pi]^l)} \right\rceil \tag{15}$$

repetitions. For any query $\boldsymbol{q} \in \mathbb{R}^r$, LSH outputs each probe vector $\boldsymbol{p} \in \boldsymbol{P}$ such that $\bar{\boldsymbol{q}}^T \bar{\boldsymbol{p}} \geq \theta$ with probability at least R.

The theorem immediately implies an expected recall of at least R.

Note that LSH has recently been applied in transformation-based methods for the MIPS problem [Shrivastava and Li 2014a; Neyshabur and Srebro 2014; Shrivastava and Li 2014b]. We discuss these methods in Sec. 7 and study their performance in Sec. 8.

5.2. LEMP with Adaptive LSH

In this section, we introduce the LEMP-LSHA algorithm, which makes use of LSH in each of LEMP's buckets. We first discuss the approximate Above-θ-MIPS problem and then proceed to Top-k-MIPS. In both cases, LEMP-LSHA takes as input a desired recall parameter R and guarantees to output each true result vector with probability at least R. The key idea of LEMP-LSHA is to use an adaptive—i.e., query- and bucket-dependent—number of LSH repetitions to ensure recall R with as low computational cost as possible.

LEMP-LSHA for Above-θ-MIPS. Assume for now that the length l of the hash code is fixed. Recall that LEMP solves many small cosine similarity search problems, one for each of its buckets, and that LEMP uses a query- and bucket-dependent local threshold $\theta_b(\boldsymbol{q})$ for cosine similarity search. Since the local threshold $\theta_b(\boldsymbol{q})$ is not constant, we cannot simply use Eq. (15) to determine the number $\mathcal{L}$ of repetitions to use to achieve recall R. The main problem is thus to obtain a suitable choice of $\mathcal{L}$ within the LEMP framework.

The idea of LEMP-LSHA is as follows. We store with each bucket b a number c_b of LSH repetitions; $c_b = 0$ initially for all buckets. When processing query $\boldsymbol{q}$ on bucket b, we compute the local threshold $\theta_b(\boldsymbol{q})$ as before. We then determine the necessary number $\mathcal{L} = \mathcal{L}(\theta_b(\boldsymbol{q}))$ of LSH repetitions needed for this bucket and query according to Eq. (15). If $c_b < \mathcal{L}$, we create $\mathcal{L} - c_b$ additional LSH repetitions by reindexing probe vectors and subsequently increase c_b accordingly. In other words, the construction of LSH repetitions is done lazily as needed. After this step, it holds $c_b \geq \mathcal{L}$, i.e., we have a sufficient number of repetitions stored with bucket b. We now pick the first $\mathcal{L}$ of these repetitions to obtain the candidate set using LSH; this step involves computing $l\mathcal{L}$ hash values of the query vector. Note that we may use less than the c_b repetitions stored with bucket b in this step: Since $\mathcal{L}$ repetitions are sufficient to achieve the desired recall, using more than $\mathcal{L}$ repetitions would be wasteful. In addition, we use the same hash functions for all buckets. This allows us to cache and re-use the signatures of the query vector when processing different buckets.

As with most methods for cosine similarity search, LSH is only effective if $\theta_b(\boldsymbol{q})$ is large. Thus we use LSH only when $\theta_b(\boldsymbol{q})$ is large and $\mathcal{L}(\theta_b(\boldsymbol{q}))$ does not exceed a pre-specified space budget. Otherwise, we use the exact NORM method. To decide whether or not to use LSH, we use the tuning method described in Sec. 4.4. We subsequently refer to the adaptive version of LSH in combination with NORM as LSHA.

The correctness of LEMP-LSHA follows immediately from its construction. We either use NORM on each bucket (providing exact results) or use a sufficient number of LSH repetitions (providing recall R).

THEOREM 5.2. *Consider the approximate Above-θ-MIPS problem and fix a recall threshold R. For each query $\boldsymbol{q} \in \boldsymbol{Q}$ and each probe vector $\boldsymbol{p} \in \boldsymbol{P}$ it holds:*

(1) *If $\mathbf{q}^T\mathbf{p} \geq \theta$, LEMP-LSHA outputs $(\boldsymbol{q}, \boldsymbol{p})$ with probability at least R.*
(2) *Otherwise, if $\mathbf{q}^T\mathbf{p} \leq \theta$, LEMP-LSHA does not output $(\boldsymbol{q}, \boldsymbol{p})$.*

PROOF. Fix $(\boldsymbol{q}, \boldsymbol{p})$. Let b be the bucket that contains $\boldsymbol{p}$. Suppose that $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$. If LEMP uses NORM on bucket b, $\boldsymbol{p}$ is included into the candidate set because NORM is an exact method. If LEMP uses LSH on bucket b, $\boldsymbol{p}$ is included with probability at least R since we use sufficiently many repetitions according to Th. 5.1. This establishes the first assertion. The second assertion holds because LEMP verifies all candidate vectors, i.e., it outputs $(\boldsymbol{q}, \boldsymbol{p})$ only if $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$. □

It remains to select the length parameter l of the hash code as well as the space budget for storing repetitions. Parameter l is usually tuned in a dataset-specific way. In our setting, however, buckets contain few vectors by construction (so that they fit into the cache). Since computing hash values requires l inner products per LSH repetition, we cannot afford to use a large value of l; otherwise, NORM would be more efficient than LSH. We thus keep l small. For similar reasons and to keep space consumption acceptable, we set the per-bucket budget of LSH repetitions to a relatively small value. In particular, our implementation fixes $l = 8$ and uses a budget of 200 repetitions; these choices provided good results across all datasets in our experimental study.

LEMP-LSHA for Top-k-MIPS. Recall from Sec. 4.5 that Top-k-MIPS for a given query vector $\boldsymbol{q}$ is equivalent to Above-θ^*-MIPS for $\theta^* = s_{(k)}$. Observe that θ^* depends on both $\boldsymbol{q}$ and $\boldsymbol{P}$ and may vary wildly across queries. When LSH is used for top-k search, θ^* is unknown, which poses severe difficulties. One way to support top-k processing is to perform a grid search to select suitable values of l and $\mathcal{L}$ empirically. Another way is to use a sequence of LSH structures with decreasing threshold values; the last LSH structure should use a threshold smaller than the smallest top-k inner product value for any query.[6] The first option does not provide any quality guarantees and is generally cumbersome and inefficient (esp. when queries have wildly varying values of θ^*). This problem has also been observed in our experimental study; see Sec. 8. The second approach is costly because the cost of signature construction is determined by the worst-case query. There is also an inherent risk of constructing too many (high preprocessing cost) or too few (lower recall than desired) of these LSH structures.

In the context of LEMP, we can avoid the problems mentioned above and derive an efficient LSH-based algorithm for Top-k-MIPS. Our algorithm uses the techniques of Sec. 4.5 on LEMP for Top-k-MIPS, but employs LSHA instead of an exact search method in each bucket. In more detail, we maintain a top-k list of the probe vectors with the largest inner products found so far; this top-k list also allows us to obtain a lower bound $\hat{\theta}$ on $\theta^* = s_{(k)}$. The top-k list is initialized with the k longest probe vectors (all at the start of bucket $\boldsymbol{P}_1$). We then process buckets in order of decreasing L2 norm. We use $\hat{\theta}$ to obtain the local threshold $\hat{\theta}_b(\boldsymbol{q})$ for the next bucket b; this local threshold is then used to obtain the candidate set from bucket b with LSHA. After candidates have been obtained, we update the top-k list and $\hat{\theta}$ and proceed to the next bucket (now with the modified value of $\hat{\theta}$). Using this approach, LEMP-LSHA avoids the problems of the plain LSH methods by frequently estimating and updating the threshold value to use.

THEOREM 5.3. *Consider the approximate Top-k-MIPS problem and fix a recall threshold R. For each query $\boldsymbol{q} \in \boldsymbol{Q}$ and each probe vector $\boldsymbol{p} \in \boldsymbol{P}$ such that $\boldsymbol{p} \in T_k(\boldsymbol{q})$, LEMP-LSHA outputs $(\boldsymbol{q}, \boldsymbol{p})$ with probability at least R.*

PROOF. Fix $(\boldsymbol{q}, \boldsymbol{p})$ and suppose that $\boldsymbol{p} \in T_k(\boldsymbol{q})$. We know that $\boldsymbol{q}^T\boldsymbol{p} \geq \theta^*$ by definition of $\theta^* = s_{(k)}$. Let b be the bucket that contains $\boldsymbol{p}$, and let $\hat{\theta}$ be the (random) threshold value that LEMP-LSHA uses to run LSHA on bucket b. Since $\hat{\theta}$ is based on the (ap-

[6]http://www.mit.edu/~andoni/LSH/manual.pdf

proximate) top-k list found so far, we must have $\hat{\theta} \leq \theta^* = s_{(k)}$. By Th. 5.2, LEMP-LSHA then includes $(\boldsymbol{q}, \boldsymbol{p})$ into the candidate set with probability at least R. Since $\boldsymbol{p} \in T_k(\boldsymbol{q})$, $\boldsymbol{p}$ will be immediately added to the approximate top-k list and not be removed later on. □

Bayesian LSH. It is conceivable to replace the basic LSH algorithm by more advanced techniques. A recent approach is BayesLSH-Lite [Satuluri and Parthasarathy 2012], which uses a Bayesian approach for candidate pruning. After gathering the candidates from the LSH bins, but before verifying them, BayesLSH-Lite employs an additional filtering step. In more detail, it determines via Bayesian inference a high-probability lower bound on the number of hash matches between the signatures of the query and a probe vector. Since inference can be costly, these bounds are precomputed for each bucket, based on a worst-case choice of $\theta_b(\boldsymbol{q})$. The additional filtering step of BayesLSH-Lite also induces some runtime overhead so that it is not immediately clear whether filtering improves performance overall. We expect to see an improvement especially when the inner product computations required for candidate verification are expensive (i.e., the dimensionality r is large). In our experimental study, we investigated the performance of BayesLSH-Lite within LEMP and found that the cost of filtering was often larger than its benefits.

5.3. LEMP-ABS and LEMP-REL for Approximate Top-k-MIPS

In this section, we describe the LEMP-ABS and LEMP-REL methods for approximate Top-k-MIPS, which provide RMSE and ARE quality guarantees, respectively. Both LEMP-ABS and LEMP-REL can be used with any exact bucket algorithm.

To see how we can use LEMP for approximate Top-k-MIPS, recall the recommender system use case and fix a user (query vector). We are interested in retrieving the top-k recommended items (probe vectors) for that user. Suppose that the ratings of these items (inner products) lie on a scale from 1 (bad) to 5 (great). The key idea of our algorithms is as follows: if the best top-k list of the user contains items with ratings between, say, 4.9–5, then an approximate top-k list with items rated, say, 4.8–5 is almost as good. If the approximate list can be be retrieved significantly faster, then this small loss in quality is acceptable: a fast good result may be preferable to a slow perfect result. This observation is exploited by LEMP-ABS and LEMP-REL: Both algorithms augment the threshold computation of LEMP so that LEMP retrieves good, but not perfect, results. In more detail, we use threshold values that are larger than the ones needed for exact Top-k-MIPS, which in turn leads to faster processing times.

The difference between LEMP-ABS and LEMP-REL lies in how the augmentation of the threshold is performed. Recall that LEMP maintains a running lower bound $\hat{\theta}$ on the optimal threshold θ^* for Top-k-MIPS. Before processing each bucket, we augment $\hat{\theta}$ based on an *error parameter* $\epsilon \geq 0$:

— LEMP-ABS augments $\hat{\theta}$ by an additive error term, i.e., we set

$$\hat{\theta}_{abs}(\epsilon) = \hat{\theta} + \epsilon. \tag{16}$$

— LEMP-REL augments $\hat{\theta}$ by a relative error term, i.e., we set

$$\hat{\theta}_{rel}(\epsilon) = \begin{cases} \hat{\theta}/(1-\epsilon) & \hat{\theta} \geq 0 \\ \hat{\theta} & \hat{\theta} < 0, \end{cases} \tag{17}$$

for $0 \leq \epsilon < 1$. Note that we do not augment $\hat{\theta}$ if it is negative (in which case the ARE may not be a meaningful measure).

We then compute the local thresholds based on $\hat{\theta}_{abs}(\epsilon)$ or $\hat{\theta}_{rel}(\epsilon)$, respectively, and proceed as in exact LEMP for Top-k-MIPS. Note that each augmented threshold value is larger than the non-augmented threshold when $\epsilon > 0$; both values are equal when $\epsilon = 0$.

The error parameter directly corresponds to quality guarantees on the obtained result. The following theorem establishes that for LEMP-ABS, ϵ is an upper bound on the RMSE of the approximate result.

THEOREM 5.4. *For any query $\boldsymbol{q} \in \boldsymbol{Q}$, the RMSE of LEMP-ABS for Top-k-MIPS with error parameter $\epsilon \geq 0$ is at most ϵ.*

PROOF. Denote by $\hat{s}_{(i)}$ the i-th largest score in the approximate top-k list obtained by LEMP-ABS. Assume for the moment that

$$\hat{s}_{(i)} + \epsilon \geq s_{(i)} \tag{18}$$

for $1 \leq i \leq k$. Then

$$RMSE = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(s_{(i)} - \hat{s}_{(i)})^2} \leq \sqrt{\frac{1}{k}\sum_{i=1}^{k}((\hat{s}_{(i)} + \epsilon) - \hat{s}_{(i)})^2} = \epsilon,$$

as desired.

It remains to show that (18) holds for all i. To see this, observe that for each of its buckets, LEMP-ABS uses a threshold $\hat{\theta}$ that satisfies $\hat{\theta} \leq \hat{s}_{(k)}$. This is because LEMP-ABS takes $\hat{\theta}$ to be the lowest inner product value in the current top-k list, which is upper bounded by its final value $\hat{s}_{(k)}$. This implies that $\hat{\theta}_{abs}(\epsilon) = \hat{\theta} + \epsilon \leq \hat{s}_{(k)} + \epsilon$ for all buckets. Denote by $\boldsymbol{p}_{(1)}, \ldots, \boldsymbol{p}_{(k)}$ the exact result of Top-k-MIPS; we have $s_{(i)} = \boldsymbol{q}^T\boldsymbol{p}_{(i)}$. Let $u \leq k$ be the largest index such that $s_{(u)} > \hat{s}_{(k)} + \epsilon$ (if such an index exists). Pick any i, $1 \leq i \leq u$, and denote by b the bucket that contains $\boldsymbol{p}_{(i)}$. Since $\hat{\theta}_{abs}(\epsilon) \leq \hat{s}_{(k)} + \epsilon$ for all buckets, including bucket b, and since $s_{(i)} > \hat{s}_{(k)} + \epsilon$ by our choice of i, LEMP-ABS will include $\boldsymbol{p}_{(i)}$ into its candidate list when processing bucket b. Since $s_{(i)}$ is among the k-th largest scores overall, $\boldsymbol{p}_{(i)}$ will subsequently be added to the running top-k list and not be evicted later on. Thus vectors $\boldsymbol{p}_{(1)}, \ldots, \boldsymbol{p}_{(u)}$ are included in the final top-k list of LEMP-ABS. For $i \leq u$, we thus have $s_{(i)} = \hat{s}_{(i)}$ so that Eq. (18) holds. Now consider any index $i > u$. We have $s_{(i)} \leq \hat{s}_{(k)} + \epsilon$ by our choice of u. Since $\hat{s}_{(k)} \leq \hat{s}_{(i)}$, it follows that $s_{(i)} \leq \hat{s}_{(i)} + \epsilon$, i.e., Eq. (18) holds. □

The error bound on the RMSE obtained by LEMP-ABS is absolute, i.e., it does not depend on the scale of the values in the actual result. In cases where the top-k values can differ wildly across different queries, it may be more appropriate to use relative error bounds instead. This means that we require small error for results with small top-k values, but allow for larger error for results with large top-k values. Such bounds are achieved by LEMP-REL.

THEOREM 5.5. *For any query $\boldsymbol{q} \in \boldsymbol{Q}$, the ARE of LEMP-REL for Top-k-MIPS with error parameter $0 \leq \epsilon < 1$ is at most ϵ.*

PROOF. Using the notation above, suppose that $\hat{s}_{(k)} < 0$ when LEMP-REL terminates. Then for all buckets, we must have had $\hat{\theta} < 0$ (since $\hat{\theta} \leq \hat{s}_{(k)}$ by definition) and thus $\hat{\theta}_{rel}(\epsilon) = \hat{\theta}$. This implies that whenever $\hat{s}_{(k)} < 0$, LEMP-REL did not augment the threshold and thus produced exact results. The ARE is thus 0 and the assertion holds.

Now suppose that $\hat{s}_{(k)} \geq 0$. Then we can show using arguments as in the proof of Th. 5.4 that

$$\hat{s}_{(i)}/(1-\epsilon) \geq s_{(i)} \tag{19}$$

for $1 \leq i \leq k$. The ARE satisfies

$$ARE = \frac{1}{k}\sum_{i=1}^{k}\frac{s_{(i)} - \hat{s}_{(i)}}{s_{(i)}} = \frac{1}{k}\sum_{i=1}^{k} 1 - \frac{\hat{s}_{(i)}}{s_{(i)}} \leq \frac{1}{k}\sum_{i=1}^{k} 1 - \frac{\hat{s}_{(i)}}{\hat{s}_{(i)}/(1-\epsilon)} = \epsilon \tag{20}$$

as asserted. □

To further improve the performance of LEMP-ABS and LEMP-REL, we use the augmented thresholds $\hat{\theta}_{abs}(\epsilon)$ and $\hat{\theta}_{rel}(\epsilon)$ only for candidate generation but not during verification. That is, we update the top-k list by taking into the consideration all candidate vectors.

5.4. Hybrid Methods for Approximate Top-k-MIPS

In this section, we explore hybrid methods that combine LEMP with both LSHA (Sec. 5.2) and threshold augmentation (Sec. 5.3) for further efficiency gains.

Recall that LEMP-ABS and LEMP-REL augment the local thresholds in order to both (1) reduce the candidate set for each each processed bucket and (2) to achieve *early termination*, i.e., stop processing further buckets even though they may contain probe vectors that improve the top-k list obtained so far. In LEMP-LSHA, on the other hand, we use LSHA to reduce the candidate set, but we do not employ early termination. Our hybrid methods combine both ideas, i.e., they use LSHA for reducing the candidate set and threshold augmentation for early termination. We refer to the resulting methods as LEMP-HYB-ABS and LEMP-HYB-REL, depending on whether we use absolute or relative threshold augmentation, respectively.

We now turn attention to the quality guarantees provided by the hybrid methods. We focus on LEMP-HYB-ABS, which uses the threshold augmentation of Eq. (16). The analysis for LEMP-HYB-REL is similar and omitted.

Fix a query $\boldsymbol{q}$ and an error parameter $\epsilon \geq 0$. As outlined in Fig. 5, we conceptually divide the probe vectors into "very good" (score $\geq s_{(k)}+\epsilon$), "good" (score in $[s_{(k)}, s_{(k)}+\epsilon)$), "quite good" (score in $[s_{(k)}-\epsilon, s_{(k)})$), and "bad" (score $< s_{(k)}-\epsilon$), where as before $s_{(k)}$ denotes the top-k score for $\boldsymbol{q}$. Observe that ϵ controls which probe vectors are considered good. Denote by the *ϵ-reduced top-k list* $T^{-}_{k,\epsilon}(\boldsymbol{q})$ the set of all probe vectors with a score of at least $s_{(k)}+\epsilon$ (very good), and by the *ϵ-extended top-k list* $T^{+}_{k,\epsilon}(\boldsymbol{q})$ the set of probe vectors with a score of at least $s_{(k)}-\epsilon$ (not bad). Note that $T^{-}_{k,\epsilon} \subseteq T_k \subseteq T^{+}_{k,\epsilon}$.

Given a recall threshold $0 < R < 1$, Th. 5.3 asserts that (1) LEMP-LSHA outputs each vector in the top-k list with probability at least R and implies that (2) the expected number of probe vectors from the true top-k list output by LEMP-LSHA is least Rk. The approximation guarantee of LEMP-HYB-ABS is similar to but—due to early termination—somewhat weaker than the ones of LEMP-LSHA. The following theorem provides some guidance about when the hybrid method is expected to work well.

THEOREM 5.6. *Consider the approximate Top-k-MIPS problem and fix a recall threshold R and an error parameter $0 \leq \epsilon < 1$. For each query $\boldsymbol{q}$,*

(1) *LEMP-HYB-ABS outputs each probe vector $\boldsymbol{p} \in T^{-}_{k,\epsilon}(\boldsymbol{q})$ with probability at least R.*
(2) *The expected number of probe vectors $\boldsymbol{p} \in T^{+}_{k,\epsilon}(\boldsymbol{q})$ output by LEMP-HYB-ABS is at least Rk.*

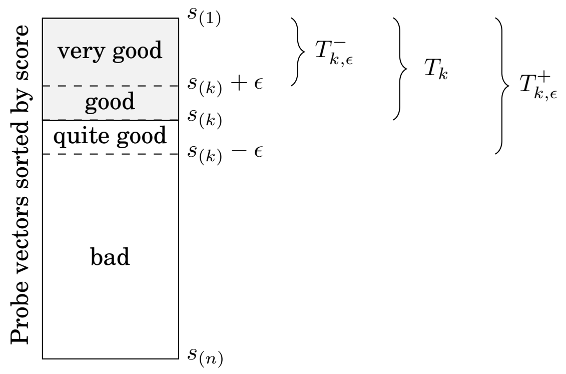


Fig. 5: Illustration of top-k list (gray), ϵ-reduced top-k list $T^-_{k,\epsilon}$ and ϵ-extended top-k list $T^+_{k,\epsilon}$ (after fixing some query $\boldsymbol{q}$)

The theorem asserts that (1) LEMP-HYB-ABS works just as well as LEMP-LSHA in retrieving very good vectors and (2) retrieves few bad vectors in expectation. For the remaining vectors, LEMP-HYB-ABS and LEMP-LSHA give different guarantees. The smaller the ϵ-extended top-k list $T^+_{k,\epsilon}(\boldsymbol{q})$, the higher the chances that the hybrid method produces a result as good as LEMP-LSHA. Put differently, if the top-k probe vectors are well separated in score from the remaining vectors, LEMP-HYB-ABS is expected to work well.

PROOF. Fix an arbitrary query $\boldsymbol{q}$. Our proof relates the output of LEMP-HYB-ABS to the output of LEMP-LSHA. We will analyze LEMP-HYB-ABS in terms of its *precision* w.r.t. to the various top-k lists. Given the output for query $\boldsymbol{q}$ of an approximate Top-k-MIPS algorithm X, denote by the random variables P^-_X, P_X, and P^+_X the set of probe vectors from $T^-_{k,\epsilon}$, T_k, and $T^+_{k,\epsilon}$, resp., in the approximate answer. Since $T^-_{k,\epsilon} \subseteq T_k \subseteq T^+_{k,\epsilon}$, we have

$$P^-_X \subseteq P_X \subseteq P^+_X. \tag{21}$$

We show below that when LEMP-LSHA and LEMP-HYB-ABS use the same set of LSH signatures, it holds

$$P^-_{HYB} = P^-_{LSHA} \tag{22}$$

and

$$|P^+_{HYB}| \geq |P_{LSHA}|. \tag{23}$$

Recall that according to Th. 5.3, LEMP-LSHA outputs each of the probe vector in T_k with probability at least R. Using this result, and since Eq. (22) states that both algorithms output the same set of vectors from $T^-_{k,\epsilon} \subseteq T_k$, our first assertion follows. Since Th. 5.3 implies $E[|P_{LSHA}|] \geq Rk$, and since $E[|P^+_{HYB}|] \geq E[|P_{LSHA}|]$ by Eq. (23), we obtain $E[|P^+_{HYB}|] \geq Rk$, our second assertion.

It remains to show that Eqs. (22) and (23) indeed hold. Fix arbitrary LSH signatures for all probe vectors as well as for $\boldsymbol{q}$. Given these signatures, LEMP-LSHA and LEMP-HYB-ABS are deterministic. Suppose that LEMP-LSHA terminates after processing l buckets. LEMP-HYB-ABS may terminate earlier, i.e., after $l' \leq l$ buckets. Observe that after processing buckets $1, \ldots, l'$, both algorithms agree in their result so far, but only LEMP-LSHA may process further buckets and thus improve the result. There are three cases:

— $l' = l$: Both methods produce the same result. Eq. (22) holds trivially. Eq. (23) follows from Eq. (21) since $P_{LSHA} = P_{HYB} \subseteq P^{+}_{HYB}$.
— $l' < l$ and buckets $l' + 1, \ldots, l$ do not contain a vector from T_k: LEMP-LSHA and LEMP-HYB output the same set of vectors from T_k, i.e., $P_{LSHA} = P_{HYB}$. Eqs. (22) and (23) follow using similar arguments as above.
— $l' < l$ and and buckets $l' + 1, \ldots, l$ contain at least one vector from T_k. Denote by $\hat{s}_{(k)}$ the smallest score output by LEMP-HYB-ABS. First observe that all probe vectors in buckets $l' + 1, \ldots, l$ have a score of less than $\hat{s}_{(k)} + \epsilon$, since otherwise LEMP-HYB-ABS would not have terminated early. In other words, all vectors that enter the approximate top-k list of LEMP-LSHA when processing buckets $l' + 1, \ldots, l$ must have a score in $[\hat{s}_{(k)}, \hat{s}_{(k)} + \epsilon)$. Since buckets $l' + 1, \ldots, l$ contain a vector from T_k, and since this vector has score at least $s_{(k)}$, we conclude that $s_{(k)} \in [\hat{s}_{(k)}, \hat{s}_{(k)} + \epsilon)$ and thus $\hat{s}_{(k)} > s_{(k)} - \epsilon$. Since $\hat{s}_{(k)}$ is so large, all vectors output by LEMP-HYB-ABS belong to $T^{+}_{k,\epsilon}$. We have $|P^{+}_{HYB}| = k \geq |P_{LSHA}|$, establishing Eq. (23). Furthermore, buckets $l' + 1, ..., l$ cannot contain a vector from $T^{-}_{k,\epsilon}$, since such a vector would have score at least $s_{(k)} + \epsilon$, but buckets $l' + 1, ..., l$ contains vectors with score less than $\hat{s}_{(k)} + \epsilon$ and thus less than $s_{(k)} + \epsilon$. Thus LEMP-LSHA and LEMP-HYB-ABS agree on the vectors output from $T^{-}_{k,\epsilon}$, establishing Eq. (22).

□

6. Parallelizing LEMP

In this section, we show how to take advantage of multithreading and instruction level parallelism for MIPS. When multiple queries are considered, the MIPS problem becomes embarrassingly parallel: queries can be partitioned among threads; each thread can run its own instance of the problem. All algorithms described in this article (LEMP, naive, cover trees [Curtin et al. 2013], TA [Fagin et al. 2001], simpleLSH [Neyshabur and Srebro 2014], PCA-Tree [Bachrach et al. 2014], etc.) can be parallelized in this way. Here we show how to apply such ideas to LEMP in an efficient way that scales to many processors.

Multithreading. When multithreading is used, it is important for scalability to a large number of cores to avoid synchronization and cache misses to the extent possible. We avoid cache misses by enforcing during bucketization the restriction that each bucket should fit into the available cache per core. Synchronization, on the other hand, can in general take place in LEMP in three places: (i) when a thread writes its results to memory, (ii) when a thread needs to obtain the next query to work on, and (iii) when a thread needs to access an index which is not yet created. To handle (i), we assign to each thread a separate memory area to write its results. The final result is then given by the union of these memory areas. To handle (2), we partition the queries among the threads during the preprocessing phase. Thus each thread knows upfront which queries it is responsible for and no further synchronization is needed. To ensure a similar workload among threads, queries are assigned to partitions randomly. We handle (iii) as follows. When a thread needs to access an index which is not yet created during the search phase, it needs to obtain exclusive access to the index in order to build it. While the index is being built, all other threads that try to read this index need to wait; we want to minimize such waiting times. For the Above-θ-MIPS problem, in which we know θ upfront, we compute in advance which buckets can contribute to the result in the worst case (longest query vector). The indexes of these buckets are created in parallel during the preprocessing phase. Thus no synchronization is needed during the search phase. For Top-k-MIPS, for which the set of required indexes is not known upfront, synchronization is inevitable. However, we can reduce synchronization overhead during search phase as follows. Recall that during tuning, we run a sample of queries against the probe buckets, so that their performance w.r.t. the NORM algorithm and other direction-based algorithms (for which we need indexes) is assessed. After we assess the performance of the sample w.r.t. NORM, we have a first crude estimate of how many buckets are contributing to the result. At this point, we pause the tuning phase, build the estimated number of required indexes for these buckets in parallel and then continue tuning. In this way, a large part of the required indexes is created before the search phase starts. If additional indexes are needed during the search phase, we build them on demand and require synchronization. Our experiments suggest that the combination of these techniques allows LEMP to scale almost linearly to a large number of processors.

Instruction-level parallelism. LEMP's performance (as well as the one of some other methods) can benefit from the use of instruction-level parallelism. As a proof of concept, we extended LEMP to use SSE instructions to speedup (i) the inner-product calculation during the verification phase, and (ii) the maintenance of the extended CP-array. Recall that while ICOORD is scanning the sorted lists, it maintains for each encountered probe vector both a partially seen inner product and an upper bound on the remaining unseen part. Maintaining these quantities involves two multiplications, which we parallelize using SIMD instructions.

7. Related Work

We first review related literature for exact MIPS and cosine similarity search, and subsequently proceed to approximate methods. In general, LEMP differs from existing methods in that it bucketizes vectors by their L2 norm, uses inexpensive bucket-level and within-bucket pruning strategies (instead of more powerful but also more expensive ones), and provides approximation guarantees for approximate MIPS.

The LEMP framework was first introduced by Teflioudi et al. [2015], who study exact retrieval in a sequential setting. This article extends the original LEMP framework as follows: (1) We study approximate MIPS and propose as well as analyze multiple novel approximate methods within the LEMP framework. (2) We discuss how to parallelize LEMP using multithreading and/or instruction-level parallelism. (3) We significantly expanded the experimental study and included results on approximate algorithms, the impact of factorization rank on the algorithms, and the effectiveness of parallelization on a large number of processors.

7.1. Exact Methods

Algorithms for MIPS. Ram and Gray [2012] address the problem of Top-k-MIPS, by organizing the probe vectors in a *metric tree*, in which each node is associated with a sphere that "covers" the probe vectors below the node. Given a query vector, the spheres are exploited to avoid processing subtrees that cannot contribute to the result. The metric tree itself is constructed by repeatedly splitting the set of probe vectors into two partitions (based on Euclidean distances). In subsequent work [Curtin et al. 2013], the metric tree is replaced a *cover tree* [Beygelzimer et al. 2006]. Both approaches effectively prune the search space, but they suffer from high tree-construction costs and from random memory access patterns during tree traversal. The latter problem was investigated more closely by Curtin and Ram [2014], who proposed a *dual-tree algorithm* that additionally arranges query vectors in a cover tree and processes queries in batches. The dual-tree method loosens the bounds for pruning the search space, however, and was found to be ineffective in practice [Curtin and Ram 2014; Teflioudi et al. 2015].

LEMP differs from these tree-based techniques in that it separates the L2 norm and direction information, makes use of multiple, light-weight indexing and search methods, and has more favorable memory access patterns. Note that the tree-based approaches can also be used within the LEMP framework as a bucket algorithm. We expect that such a combination will have positive effect w.r.t. indexing time and cache locality. We explored this approach in our experimental study.

An alternative approach is taken by Zhang et al. [2014] in the context of recommender systems: the matrix factorization method used to produce the input matrices is modified such that all vectors are (approximately) unit vectors. Then inner products between user and item vectors can be approximated by their cosine similarity, which enables the use of existing methods for cosine-similarity search. This modification may affect the quality of the recommendations, however, and is not suitable for all applications. In contrast, LEMP makes no assumption on the source or method used to compute the input matrices.

Threshold algorithm. Some of the indexing techniques used in LEMP are inspired by the popular threshold algorithm (TA) of Fagin et al. [2001] for top-k query processing for monotonic functions. TA arranges the values of each coordinate of the probe vectors in a sorted list, one per coordinate. Given a query, TA repeatedly selects a suitable list (e.g., round robin or heap-based), retrieves the next vector from the top of the list, and maintains the set of the top-k results seen so far. TA uses a termination criterion to stop processing as early as possible. The effectiveness of this criterion depends on the data; if TA is able to stop early, it can be very efficient. Note that TA usually focuses on vectors of low dimensionality (say up to 10), whereas we focus on vectors of medium sizes (say 10 to 500). TA can be used for finding vectors with large inner products almost as is; the only difference is that sorted lists need to be processed bottom-to-top when the respective coordinate of the query vector is negative.

LEMP improves over TA in multiple ways: First, bucket pruning eliminates early all short probe vectors, which otherwise TA would have to consider. Second, TA scans lists from top-to-bottom, whereas LEMP considers only the feasible region. Third, TA immediately computes the inner product of each vector selected from one of the lists in the index; i.e., candidate verification is triggered by individual coordinates. LEMP does not immediately calculate an inner product when it encounters a vector: it first scans multiple lists and prunes the vectors before verification based on the so-obtained information. Finally, index scan and verification is interleaved in TA, resulting in a random memory access pattern and a potentially high cache-miss rate. LEMP ensures that all bucket-related data (original vectors and indexes) fits into cache, thereby reducing the cache-miss rate.

In our experimental study we investigated the performance of TA in comparison to LEMP. We also experimented with TA in combination with LEMP, i.e., we used TA as a bucket algorithm. This addresses the first and the final point in the discussion above. Our experimental results indicate that a combination of TA and LEMP can be up to 25x faster than just using TA. Generally, LEMP can improve TA's performance for top-k problems with linear scoring functions (i.e., inner products).

Cosine similarity search. Exact cosine similarity search algorithms, like all-pairs similarity search (APSS, [Bayardo et al. 2007; Chaudhuri et al. 2006; Lee et al. 2010; Xiao et al. 2008]), cannot be used directly for the MIPS problem. However, these methods can be used (with some modifications) as search methods for LEMP's buckets or together with transformation-based MIPS methods.

Typical APSS algorithms and applications involve sparse vectors of high dimensionality (tens or hundreds of thousands of coordinates). In such settings, sparsity must be retained during indexing to keep the index size manageable. Thus APSS algorithms generally index only the non-zero values of each coordinate (in contrast to LEMP). In addition, coordinates are often permuted such that dense coordinates (called prefix) appear before sparser coordinates (suffix); only the suffix is indexed. The index is used to obtain candidate vectors, which are further pruned based on properties of prefixes and suffixes [Xiao et al. 2008; Lee et al. 2010; Anastasiu and Karypis 2014]. Finally, full similarity scores are computed for each candidate.

L2AP [Anastasiu and Karypis 2014] is a state-of-the-art APSS algorithm for exact cosine similarity search targeting vectors with real-valued coordinates. It exploits the Euclidean norms of suffixes and prefixes for index compression and candidate filtering. L2AP can be used as a bucket algorithm for LEMP after a few modifications. In particular, we create a separate L2AP index for each bucket. In L2AP, like in most APSS algorithms, a lower bound on the cosine similarity threshold of the query needs to be fixed a priori. In our setting, we pick the lower bound $\theta_b(\boldsymbol{q}_{max})$, where $\boldsymbol{q}_{max}$ is the query vector with the largest L2 norm.

L2AP follows a similar pruning technique to ICOORD during candidate generation and verification: it accumulates $\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F$ and precomputes $u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F)$. ICOORD differs in the following ways: (i) L2AP scans all indexed lists corresponding to non-zero query coordinates, whereas ICOORD scans only ϕ of them and only their feasible regions, (ii) L2AP uses sophisticated filtering conditions both during and after scanning. These filtering techniques eliminate the majority of the candidates, but are generally expensive. In contrast, ICOORD filters candidates only once and after index scanning, which is cheap but may result in a larger number of candidates. See Sec. 8 for an experimental comparison of the two methods.

7.2. Approximate Methods

Hardness. Ahle et al. [2016] studied the complexity of the MIPS problem and some of its variants. One of their main results is that Above-θ-MIPS and Top-k-MIPS are both hard to approximate in subquadratic time, assuming the strong exponential time hypothesis. In practice, however, experiments suggest that approximate methods often do work well.

Query clustering. Koenigstein et al. [2012] approached the approximate Top-k-MIPS problem by clustering the query vectors and solve the Top-k-MIPS problem only for the cluster centroids. The results for the centroids are taken as approximate results for all the queries in the respective cluster. The authors derive relative error bounds (ARE) on the results, based on the cosine similarity of the query and the centroid. If this bound is larger than a desired value, the algorithm falls back to exact search for that specific query. Query clustering can be directly applied in combination with LEMP (or any other Top-k-MIPS algorithm). We do not consider such an approach here because it was outperformed by PCA-Trees in previous studies (see below), the clustering phase may be expensive, the method's performance heavily depends on the quality and number of the clusters, and the approach is not suitable for online processing (since all queries need to be known in advance).

Transformation-based methods. Recently, a number of novel methods have been proposed that perform transformations of the query and/or probe vectors such that Top-k-MIPS is reduced to NNS [Bachrach et al. 2014; Shrivastava and Li 2014a] or to CSS [Shrivastava and Li 2014b; Neyshabur and Srebro 2014] on the transformed vectors. The existence of such transformations is promising because they enable the direct use of existing methods for NNS or CSS. All transformations slightly increase the dimensionality of the data vectors, either by one [Bachrach et al. 2014; Neyshabur and Srebro 2014] or two [Shrivastava and Li 2014a; 2014b]. The additional coordinates generally hold information related to the L2 norm of the vector. LEMP differs from these methods in that it exploits L2 norm information directly via bucketization into many small problems, rather than indirectly via transformation into one large problem. This approach allows LEMP to perform quick initial norm-based pruning for many buckets and to select suitable search algorithms for the remaining buckets.

Transformation to NNS. Bachrach et al. [2014] showed how to reduce the Top-k-MIPS to an equivalent nearest-neighbor problem in Euclidean space by introducing the following asymmetric transformations for the query and probe vectors, respectively:

$$\begin{aligned} t_{NNS}(\boldsymbol{q}) &= \begin{pmatrix} 0 & \boldsymbol{q}^T \end{pmatrix}^T \\ t_{NNS}(\boldsymbol{p}) &= \begin{pmatrix} \sqrt{(\max_i \|\boldsymbol{p}_i\|)^2 - \|\boldsymbol{p}\|^2} & \boldsymbol{p}^T \end{pmatrix}^T . \end{aligned}$$

Note that the added coordinate in $t_{NNS}(\boldsymbol{p})$ is large (and often dominant) for short probe vectors and small for long probe vectors.

After the vectors get transformed with t_{NNS}, Bachrach et al. [2014] propose the use of a so-called PCA-Tree, an approximate method, for NNS. The PCA-Tree is a binary tree of depth $d \ll r + 1$, which is formed by splitting probe vectors based on their first d principal components. Probe vectors are partitioned across the leaves of the tree. During query processing, the tree is traversed to find a set of d leaves (and the corresponding probe vectors) which best match the transformed query. The input parameter d controls the trade-off between speedup and quality: The larger d, the fewer total number of candidates, the larger the speedup, and the lower the quality of the result. The PCA-Tree method does not provide any error bounds, but was empirically shown to outperform the clustering method of Koenigstein et al. [2012].

An alternative to the PCA-Tree is to use prior state-of-the-art methods for approximate nearest neighbor search such as Annoy,[7] a popular library developed at Spotify. To the best of our knowledge, this direction has not been explored experimentally before (but we do so in our experimental study). Annoy builds a forest of binary trees, each using random hyperplanes for splitting. During query processing, Annoy maintains a priority queue with the most promising leaves (based on heuristics) to be visited, and verifies the vectors in those leaves. The number of trees and the number of leaves to be visited are parameters that control the speedup-quality tradeoff. No approximation guarantees are provided.

Another direction to approach approximate MIPS is via LSH-based methods. It is known that LSH cannot be used to solve MIPS on the original vectors [Neyshabur and Srebro 2014; Shrivastava and Li 2014a]. Shrivastava and Li [2014a] derived a transformation related to t_{NNS} and applied LSH methods for Euclidean NNS.

Transformation to CSS. In their subsequent work, Shrivastava and Li [2014b] proposed an alternative asymmetric transformation to CSS for use with LSH, which provided better results. Neyshabur and Srebro [2014] proposed another, experimentally superior transformation from Top-k-MIPS to CSS given by:

$$\begin{aligned} t_{CSS}(\boldsymbol{q}) &= \begin{pmatrix} 0 & \bar{\boldsymbol{q}}^T \end{pmatrix}^T \\ t_{CSS}(\boldsymbol{p}) &= \left(\sqrt{1 - \|\boldsymbol{p}\|^2 / \max_i \|\boldsymbol{p}_i\|^2} \;\; \boldsymbol{p}^T / \max_i \|\boldsymbol{p}_i\| \right)^T . \end{aligned}$$

Note that t_{NNS} and t_{CSS} are closely related. In fact, if we rescale $\boldsymbol{P}$ by $1/\max_i \|\boldsymbol{p}_i\|$ upfront and normalize all query vectors, t_{NNS} and t_{CSS} agree, and the result of Top-k-MIPS remains unaffected. Ahle et al. [2016] suggested to use t_{CSS} with the

[7]https://github.com/spotify/annoy

LSH method FALCONN[8] of Andoni et al. [2015]—instead of the random hyperplanes method used by Neyshabur and Srebro [2014]—for improved results.

In Section 8, we experimentally investigate the performance of transformation-based methods using t_{NNS} or t_{CSS} in combination with various state-of-the-art similarity search methods.

8. Experimental Study

We conducted an extensive experimental study using multiple real-world datasets. The goals and results of our experimental study are summarized below:

— We investigated the performance of various state-of-the-art methods for exact MIPS: naive search (Naive), LEMP-based methods, the threshold algorithm (TA), and the single and dual cover tree approaches (Tree, D-Tree). We found that LEMP consistently outperformed alternative exact methods and was the best-performing method overall. In particular, LEMP was up to multiple orders of magnitude faster than Naive and between 2x and 20x faster than the best-performing alternative method.

— We studied the relative performance of different bucket-search methods for exact MIPS with LEMP, including NORM, COORD, ICOORD, TA, cover trees, and L2AP. We found that a combination of NORM and ICOORD was the most efficient bucket-search method overall.

— We investigated the effect of the dimensionality r of the input vectors on each algorithm's performance for exact MIPS. Our results suggest that LEMP maintains its performance advantage across all dimensionalities we considered.

— We investigated the quality-speed trade-off of the LEMP-based methods as well as various transformation-based methods paired with state-of-the-art similarity search algorithms. We found that the best-performing method was dataset-dependent. On the three real-world datasets we considered for approximate MIPS, the best-performing methods were LEMP-HYB-REL, LEMP-REL, and the t_{CSS}-transformation with Annoy for similarity search, respectively. LEMP was up to roughly 4x faster than the closest alternative method not based on LEMP (for the same quality level).

— Finally, we studied the scalability of our parallel LEMP variants. We observed nearly linear speedups up to 32 processors (the largest number considered in our experiments) and a runtime decrease of up to 23% when SIMD instructions were used.

8.1. Experimental Setup

All datasets and our source code can be found at http://dws.informatik.uni-mannheim.de/en/resources/software/lemp.

Hardware. Our experiments were run on a machine with 48 GB RAM and an Intel Xeon 2.40GHz processor. Unless stated otherwise, our experiments were carried out on a single thread and no SIMD instructions were used.

Datasets. We used real-world datasets from collaborative filtering and information extraction applications (cf. Section 2.2) as well as a word-vector dataset previously used for benchmarking approximate NNS algorithms. Table I summarizes our datasets. The table gives the sizes of the input data and for various choices of rank r, the coefficient of variation (CoV) of the L2 norms of the input vectors, the percentage of non-zero entries and the time required by Naive.

[8]https://github.com/FALCONN-LIB/FALCONN

Table I: Overview of datasets

Dataset	m	n	r	CoV of L2 norms $\boldsymbol{Q}$	$\boldsymbol{P}$	% Non-Zero	Naive (min)
IE-NMF	771K	132K	10	2.05	5.49	28.2	27.1
			50	1.56	5.53	36.2	113.6
			100	1.34	4.45	50.8	244.4
IE-SVD	771K	132K	10	2.04	5.46	100	27.1
			50	1.51	4.44	100	113.6
			100	1.28	3.64	100	244.4
Netflix	480K	17K	10	0.12	0.16	100	2.0
			50	0.16	0.22	100	8.5
			100	0.19	0.22	100	20.4
KDD	1000K	624K	51	0.38	0.40	100	838.3
GloVe	100K	1.09M	100	0.20	0.20	100	290.4

For our experiments with collaborative filtering data, we used factorizations of the popular Netflix [Bennett and Lanning 2007] and KDD [Dror et al. 2012] datasets.[9] Both datasets consist of ratings of users for movies (Netflix) or musical pieces (KDD). For Netflix, we performed a plain matrix factorization with DSGD++ using L2 regularization with regularization parameter $\lambda = 50$, as in [Teflioudi et al. 2012]. For KDD, we used the factorization of Koenigstein et al. [2011],[10] which incorporates the music taxonomy, temporal effects, as well as user and item biases; this dataset has been used in previous studies of the Top-k-MIPS problem. Since we were ultimately interested in retrieving the top-k movies/songs for each user, we used the collaborative filtering datasets to study the performance of the various methods for the Top-k-MIPS problem.

For the open information extraction scenario, we extracted around 16M subject-pattern-object triples from the New York Times corpus,[11] which contains news articles, using the methods described in Nakashole et al. [2012]. We removed infrequent arguments and patterns, and constructed a binary argument-pattern matrix: An entry in the matrix was set to 1 if the corresponding argument (subject-object pair) occurred with the corresponding pattern; otherwise, the entry was set to 0. We factorized this binary matrix using the truncated singular-value decomposition (SVD) and non-negative matrix factorization (NMF); we denote the resulting datasets as IE-SVD and IE-NMF, respectively. For SVD, which produces factorization $\boldsymbol{U}_r \boldsymbol{\Sigma_r} \boldsymbol{V}_r^T$, we set $\boldsymbol{Q}^T = \boldsymbol{U}_r \sqrt{\boldsymbol{\Sigma_r}}$ and $\boldsymbol{P} = \sqrt{\boldsymbol{\Sigma_r}} \boldsymbol{V}_r^T$. For the IE datasets, we studied Above-θ-MIPS and Top-k-MIPS, which are both relevant in applications. Above-θ-MIPS aims to find all high-confidence facts, whereas Top-k-MIPS retrieves the k most probable arguments of a pattern (as in Riedel et al. [2013]). For the latter problem, we make use of the transposed matrices IE-SVD^T and IE-NMF^T.

We factorized Netflix, IE-SVD and IE-NMF with ranks 10, 50 and 100. Unless stated otherwise, we use rank $r = 50$. We investigate the effect of other choices in Section 8.3. As stated previously, fast and scalable matrix factorization algorithms have been proposed in the literature so that the time for matrix factorization is often not a bottleneck in applications. For example, we obtained IE-SVD ($r = 50$) and IE-NMF ($r = 50$) in less

[9] The KDD (Yahoo! Music) dataset corresponds to Track 1 of the 2011 KDD-Cup.
[10] We zeroed out all subnormal numbers.
[11] http://catalog.ldc.upenn.edu/LDC2008T19

than four minutes each using Matlab. As another example, Makari et al. [2015] reports that the KDD dataset ($r = 50$) can be factored in roughly seven minutes. In all three cases, the factorization time is significantly smaller than the time required to perform MIPS using Naive so that MIPS is the main bottleneck.

The GloVe[12] dataset contains 100-dimensional vector representations of words [Pennington et al. 2014]. We randomly chose 100K vectors as queries and used the remaining vectors as probe vectors. GloVe has been used in the past as a benchmark dataset for approximate NNS and CSS.[13]. Here we use the dataset for experimentally exploring Top-k-MIPS methods, i.e., for a different problem.

Exact algorithms. We considered the following exact MIPS algorithms: naive search (Naive), LEMP-based methods, the threshold algorithm (TA), and the single (Tree) and dual (D-Tree) cover tree approaches. We implemented Naive, LEMP, and TA in C++. A C++ implementation of Tree and D-Tree was provided to us by the authors of [Curtin et al. 2013; Curtin and Ram 2014].[14]

For TA, we experimented with two different list-selection schedules: a round robin (RR) on the lists corresponding to non-zero query coordinates and one that selects the sorted list i that maximized $q_i p_i$, where p_i refers to the next coordinate value in list i. The latter strategy selects the "most-promising" coordinate; we implemented it efficiently using a max-heap. We additionally improve the performance of TA by allowing multiple steps on the same list if all these steps access probe vectors that are already explored. Finally, we report the best results achieved by TA (RR or heap-based).

We ran six "pure" versions of LEMP for exact MIPS, in which only one method was used within a bucket. We denote these methods as LEMP-X, where X is: N (for NORM), C (for COORD), I (for ICOORD), TA, L2AP or Tree. Here TA, L2AP, and Tree are prior methods. We integrated the state-of-the-art CSS method L2AP into LEMP by adjusting the publicly available C code.[15]

We also ran the two mixed versions LEMP-NC (for NORM and COORD) and LEMP-NI (for NORM and ICOORD), in which the bucket-search method is automatically chosen as described in Section 4.4.

Unless stated otherwise, we use LEMP-NI as our default method and denote it by LEMP.

Approximate algorithms. We considered various transformation-based methods paired with state-of-the-art NNS or CSS algorithms for approximate MIPS. We considered the t_{NNS}-transformation with PCA-Trees, the t_{CSS}-transformation with Annoy[16], and the t_{CSS}-transformation with FALCONN.[17] We implemented PCA-Trees in C++. We also experimented with alternative transformations for Annoy and FALCONN, but consistently obtained worse results than with t_{CSS}. Similarly, FALCONN (using cross-polytope LSH and multi-probing) performed better than random hyperplane LSH.

Additionally, we used the following LEMP-based methods: LEMP-LSHA, LEMP-LSHA(B) (using BayesLSH-Lite), LEMP-REL, LEMP-HYB-REL, LEMP-HYB(B)-REL (using BayesLSH-Lite). We also experimented with LEMP-ABS and LEMP-HYB-ABS, but we omit the results here, because they were qualitatively similar to those of LEMP-REL and LEMP-HYB-REL, respectively.

[12]http://nlp.stanford.edu/projects/glove/
[13]https://github.com/erikbern/ann-benchmarks
[14]http://mlpack.org/
[15] http://glaros.dtc.umn.edu/gkhome/l2ap/overview
[16]https://github.com/spotify/annoy
[17]https://github.com/FALCONN-LIB/FALCONN

Methodology. We considered both Above-θ-MIPS and Top-k-MIPS. For Above-θ-MIPS, we selected θ such 10^3, 10^4, 10^5, 10^6, or 10^7 entries were retrieved from $\boldsymbol{Q}^T\boldsymbol{P}$. We subsequently refer to the number of results as the *retrieval level* and report the *retrieval level* instead of the actual value of θ. For Top-k-MIPS, we experimented with $k \in \{1, 5, 10, 50\}$.

Unless otherwise stated, we compare all methods in terms of overall wall-clock time, which includes preprocessing, tuning, and retrieval time. Preprocessing involves the construction of indexes and, for LEMP only, the time required for the normalization, sorting, and bucketization of the input vectors. Tuning refers to the time required to automatically select suitable values for the parameters ϕ and t_b of LEMP.

Choice of parameters. LEMP's parameters (ϕ and t_b) were tuned on a small sample of the datasets as explained in Section 4.4; tuning time is included in the runtime. The base parameter of the cover trees was set to 1.3 as suggested by Curtin and Ram [2014]. For all LEMP algorithms, we used a fine-grained bucketization such that all data structures of a bucket fit into the available processor cache. For LEMP-L2AP, we used the same combination of filters and bounds that Anastasiu and Karypis [2014] report as most efficient w.r.t. execution time. For LEMP-LSHA, we set the signature length to 8 bits and maximum number of signatures to be used to 200.

8.2. Exact MIPS

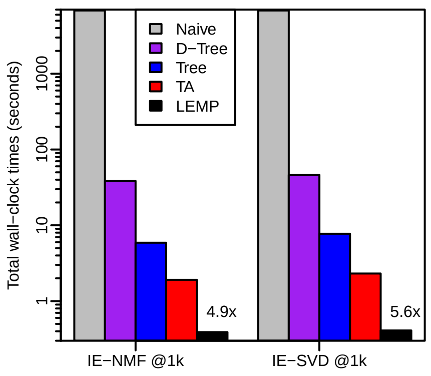
In this section, we compare LEMP with previous methods for exact MIPS. Figures 6 and 7 show the relative performance of LEMP (using the NI bucket algorithm), TA, Tree and D-Tree for the Above-θ-MIPS and Top-k-MIPS problems, respectively. The speedup of LEMP with respect to the best-performing method other than LEMP is printed in the figures. We use Naive as a baseline; its running time is independent of θ for Above-θ-MIPS and only slightly affected by k for Top-k-MIPS. To keep our study manageable, we only ran Naive for the Top-1-MIPS problem; this is a fair comparison because running times for larger k may be slightly above, but not below the times reported here. The wall-clock times for this and additional experiments, as well as average candidate set sizes, can be found in Tables IV and V in the online Appendix B accessible in the ACM Digital Library.

In the following, we discuss the performance of the algorithms in terms of overall running time, preprocessing time and pruning power.

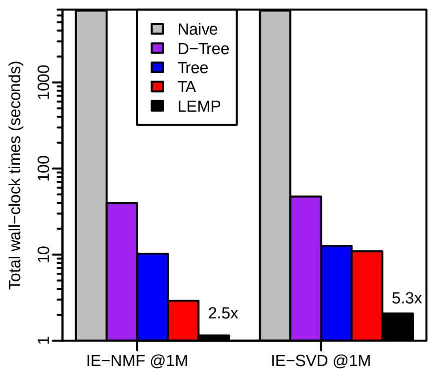
Overall performance. In general, LEMP was the fastest method, reaching up to 17000x speedup over the Naive baseline and up to 24x speedup over the next best method. The second fastest method in the majority of cases was Tree, followed by TA and D-Tree. LEMP, Tree, and TA appear to have best performance on datasets with large skew in their L2 norm distribution, like the IE datasets (high CoV in Table I) and also on datasets with sparse vectors (IE-NMF). On datasets with little skew in their L2 norm distribution, like Netflix, GloVe and KDD, all methods had difficulties in providing large speedups over Naive. However, for KDD, some methods were still able to offer significant savings in terms of running time: e.g., LEMP took 9.4 hours less time than Naive. Tables IV and V show that the performance of all methods but Naive deteriorates as the result size or k increases (θ decreases), since the output size increases and pruning opportunities decrease. Generally, there is a break-even point at which any method will be slower than Naive. This is the case, for example, for all methods other than LEMP on Netflix/GloVe/KDD for $k \geq 1$.

Preprocessing time. Table II shows the preprocessing time for the different datasets and methods.

For Tree and D-Tree, we give the wall-clock time of producing the cover tree(s) and for TA the time to create the sorted lists. The preprocessing costs of these methods are



(a) Above-θ-MIPS @1K



(b) Above-θ-MIPS @1M

Fig. 6: Total wall-clock times (incl. indexing and tuning) for exact Above-θ-MIPS on different datasets for θ values resulting to 1K and 1M results

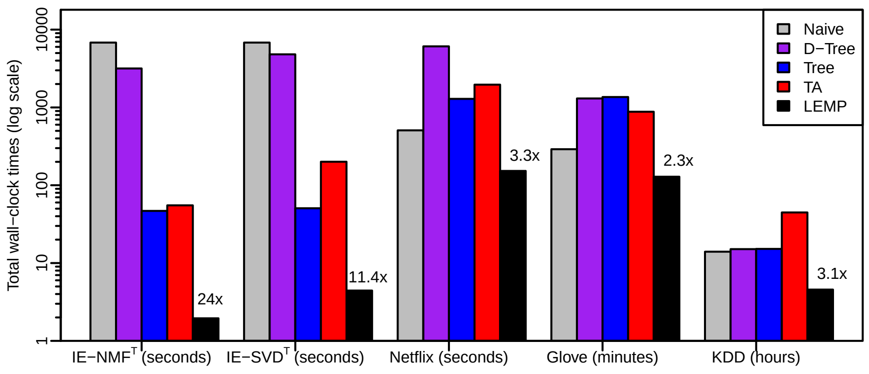


Fig. 7: Total wall-clock times (incl. indexing and tuning) for exact Top-1-MIPS on different datasets

fixed and depend on the size of probe matrix (and additionally of the query matrix for D-Tree).

For LEMP, we report the sum of maximum indexing and the maximum tuning time (normally the preprocessing times vary from problem to problem since LEMP constructs indexes lazily). On the one hand, LEMP suffers from tuning overhead, but on the other hand, it benefits from lazy index construction, especially for datasets with skewed L2 norm distribution. The larger the L2 norm skew and the size of the probe matrix, the larger the preprocessing savings of LEMP over the other methods, and the higher the chances of outweighing the tuning overhead. For example, for IE-NMF^T, which has $n = 771K$, LEMP needed 0.84s vs. 2.98s for TA and 31.84s for Tree.

The highest costs appeared for the Tree and D-Tree methods because the tree construction involves many Euclidean distance calculations between vectors. Preprocessing costs can be one of the major bottlenecks for these methods: Tables IV and V show that preprocessing can be a large part of the overall running time. For example, D-Tree

Table II: Maximum preprocessing times (in seconds) including indexing and tuning

Dataset	LEMP	TA	Tree	Dual Tree
IE-NMF	0.78	0.46	5.33	38.5
IE-SVD	1.18	0.76	7.1	46.2
IE-NMF^T	0.84	2.98	31.84	38.5
IE-SVD^T	1.51	4.88	37.5	46.4
Netflix	1.63	0.10	1.10	3510.9
KDD	63.32	4.47	208.7	2880.0
GloVe	7.92	13.86	35881.2	36373.4

took longer to index Netflix (80% of overall time) and GloVe (45%) than Naive needed to retrieve the Top-1-MIPS per query. Similarly, on the IE datasets with retrieval levels $\leq 10^6$ or $k \leq 10$, LEMP terminated before Tree finished preprocessing.

Pruning power. Tables IV and V show how many candidates remain on average after pruning for each of the different methods. For the Top-k-MIPS problem, LEMP had the highest pruning power for the IE datasets and Netflix. Note that LEMP was the only method outperforming Naive on Netflix. In fact, it is difficult to improve on Naive on this dataset: Netflix has the smallest L2 norm skew, which makes pruning less effective, and a relatively small probe matrix, which makes Naive perform reasonably well.

TA ranked often third in terms of pruning power. Especially for datasets with low L2 norm skew, TA tended to perform poorly. For example, for Netflix, $k = 1$, TA had almost no pruning power (16K candidates per query, out of a total of 17.7K). We also see the effect of TA's random memory access pattern here. Although TA verified almost the same number of candidates as Naive, it was 5.8x slower (1961.8s vs. 335.8s). Its behavior on GloVe was similar. Also note that sparsity affects the behavior of TA: It checked 3.2x less candidates for the sparse IE-NMF^T dataset, $k = 1$, than for IE-SVD^T (1899 vs. 6090 candidates per query). The main reason for the relatively low pruning power of TA for dense datasets is that it is L2 norm-oblivious, i.e., it checks short probe vectors if they have a single, sufficiently large coordinate; these vectors are discarded by LEMP. On the other hand, for sparse datasets, large values for individual coordinates correlate well with the L2 norm of the vectors so that essentially TA explores long vectors first. We expect that a combination of LEMP and TA can address the problems of L2 norm-obliviousness and random memory accesses; this approach is explored in Section 8.5.

D-Tree's query grouping generally helped to reduce the frequency of visits of the probe-tree nodes (and thus the time for candidate checking). D-Tree pruned more candidates than any other other method for Above-θ-MIPS. For Top-k-MIPS, grouping was less effective because the bounds depend on the lower bound $\hat{\theta}$ among all queries of the group, which may vary wildly and thus reduces pruning power.

Cache exploitation. Recall that LEMP does not create buckets that exceed the cache size. To study the effect of this approach, we experimented with a cache-oblivious version of LEMP in which bucket sizes were unrestricted. We found that for datasets with large L2 norm skew, runtime differences were marginal: LEMP creates small buckets anyway when L2 norms are skewed. For datasets with less L2 norm skew, such as KDD, there was a significant difference in runtime: LEMP created roughly 15x more buckets than its cache-oblivious version (26 vs. 403), and was almost 40% faster (7.9h vs. 4.56h).

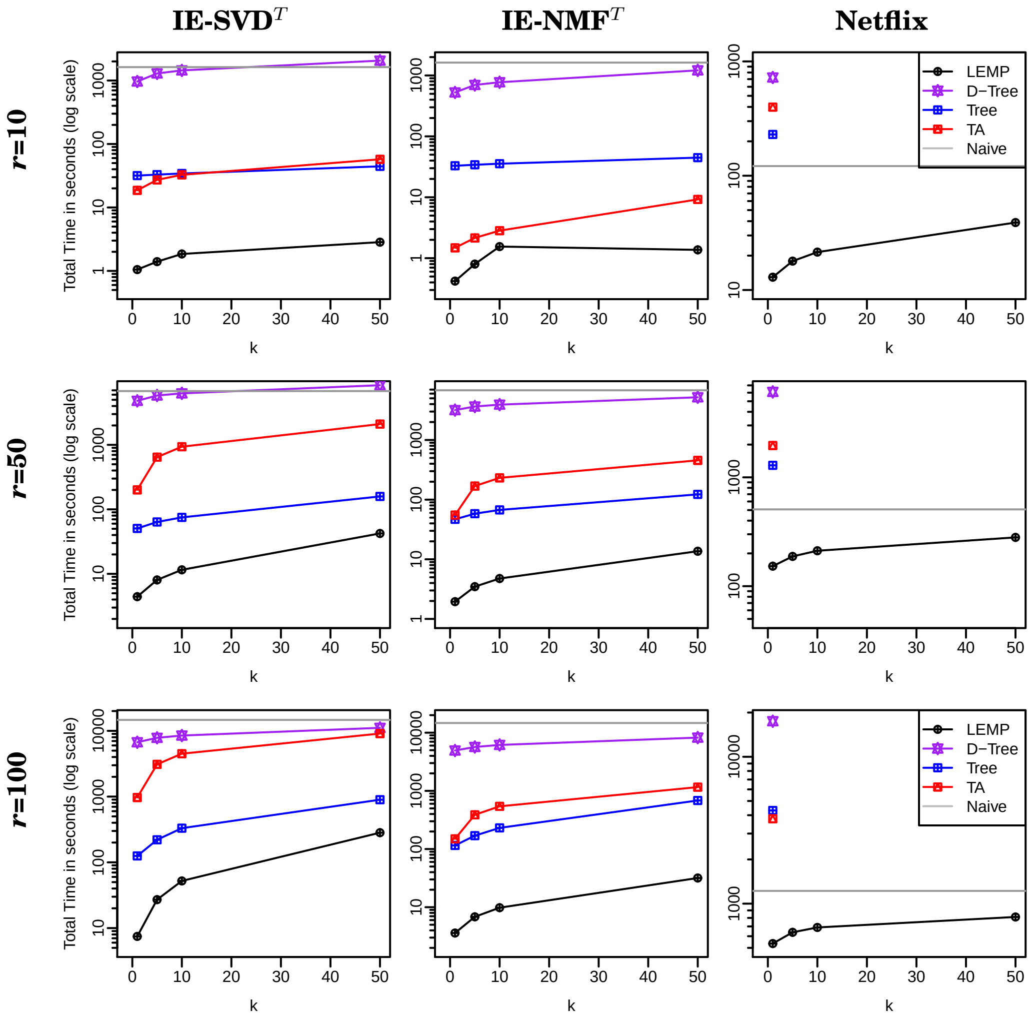


Fig. 8: Total wall-clock times for different ranks for exact Top-k-MIPS

8.3. Influence of Dimensionality

In our next experiment, we investigated the impact of the dimensionality r of the input vectors (the rank of the factorization) on the performance of exact MIPS algorithms.[18] We experimented with ranks 10, 50 and 100. The properties of the resulting datasets are summarized in Table I.

Figure 8 shows the performance (in log scale) of Tree, D-Tree, TA, LEMP (again, using LEMP-NI) and Naive for the Top-k-MIPS problem for $r \in \{ 10, 50, 100 \}$ and various values of k. We omit results for each method when it performed worse than Naive. As expected, all methods became slower when the rank was increased. Also note that the lower the rank the more competitive Naive becomes. This is because lower rank implies less expensive inner product computations and thus less work for Naive. LEMP

[18]We omit the KDD and GloVe datasets from this set of experiments because they were not available to us with dimensionality $r = 10$

was the best performing method for IE-SVD^T and IE-NMF^T regardless of the rank. TA behaved better for low ranks than for larger ranks (recall that TA is designed for vectors of low dimensionality), especially for sparse datasets (IE-NMF^T). Tree ranked almost always in between LEMP and TA, whereas D-Tree was the worst performing method.

For Netflix, all methods perform poorly, mainly because the Netflix dataset is relatively small. However, LEMP was the only method able to offer speedup over Naive.

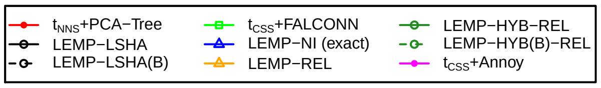
8.4. Approximate MIPS

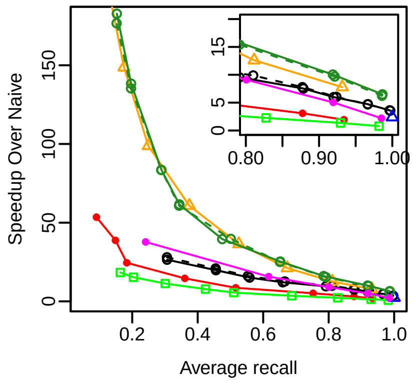
In this section, we study the performance of various methods for approximate Top-k-MIPS. Figure 9 summarizes our results for the Netflix, GloVe, and KDD datasets for $k = 10$. The runtime of the best exact method (LEMP-NI) is marked as a comparison point in the plots.

Methods. In addition to the LEMP-based methods, we study the following transformation-based methods, each paired with a state-of-the-art similarity search algorithm: the t_{NNS} transformation with PCA-Trees, the t_{CSS} transformation with the cross-polytope LSH method FALCONN, and t_{CSS} with Annoy. As mentioned previously, we also experimented with different combinations of transformations and search methods, as well as with hyperplane LSH, but the results were always worse that the combinations given above. To ensure fair comparison, we disabled SIMD instructions for all methods and used always a single core. For LEMP-REL, we used NI as the exact bucket-search algorithm.

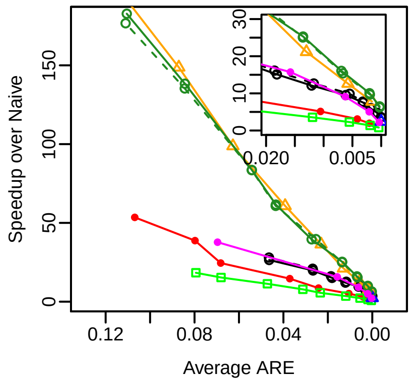
Parameterization. Each considered method provides parameters to control the speed-quality tradeoff. Each data point in Figure 9 corresponds to one setting of these parameters and indicates the resulting speedup over Naive (y-axis) as well as the value of one of our error measures (x-axis). For PCA-Trees, we varied the depth parameter d in the range 2–10. For Annoy, we performed a grid search for the best performing parameters (number of trees and number of leaves to be searched) following the example of https://github.com/erikbern/ann-benchmarks. For FALCONN, we used number of hash functions per repetition close to $\log_2 n$ as suggested, and performed a grid search on the number of hash tables and on the number of probes (the library allows for multiprobe). We present the Annoy and FALCONN results for the *best combination* of their parameters per dataset and recall value. Note that our dataset-dependent parameterization gives Annoy and FALCONN an advantage over all other methods. For LEMP-LSHA, we varied the recall parameter R in a range from 0.9–0.1 in steps of 0.1. For LEMP-REL, we varied ϵ in the range 0.2–0.7 with steps of 0.05. For LEMP-HYB-REL, we varied simultaneously R in a range from 0.9–0.1 with steps of 0.1 and ϵ in the range 0.2–0.6 with steps of 0.05.

The t_{CSS} transformation. The LSH-based approach t_{CSS}+FALCONN did not provide competitive results in our study. On Netflix and GloVe, the method was among the slowest alternatives. For Netflix, the probe matrix is small and Naive quite fast already. To be competitive, LSH cannot afford a large number of repetitions (best choice: 5–10) in order to compute query signatures fast (still 45% of the total time at around 55% recall). To gain further insight, we determined the average cosine similarities between 1000 randomly selected transformed probe vectors. For Netflix and GloVe, the average similarity was ≈ 0.9, for KDD it was ≈ 0.8. High values indicate that transformed probe vectors tend to be closer, which makes top-k similarity search more difficult. Thus transformation-based methods tended to perform better on KDD than on the other two datasets.

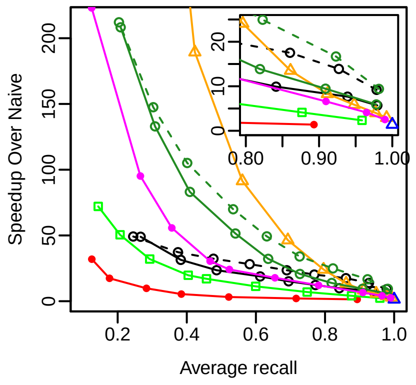




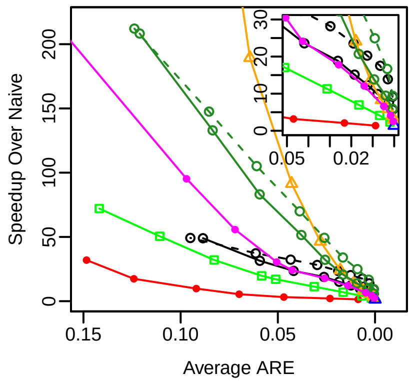
(a) Recall on Netflix



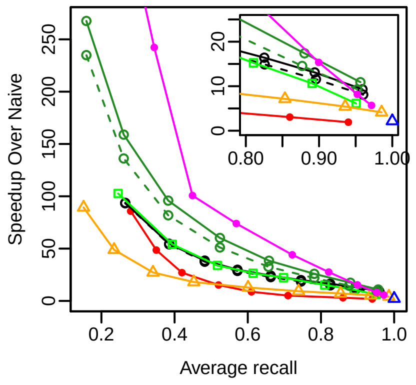
(b) ARE on Netflix



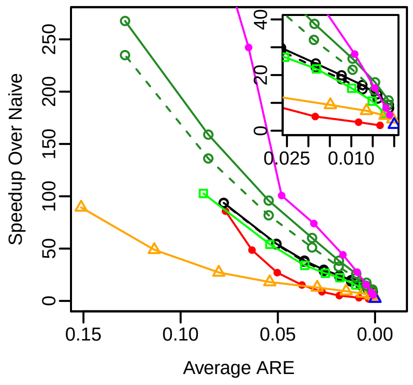
(c) Recall on GloVe



(d) ARE on GloVe



(e) Recall on KDD



(f) ARE on KDD

Fig. 9: Speedup over Naive for approximate Top-10-MIPS (higher is better).

Annoy generally outperformed LSH-based similarity search with t_{CSS}. In fact, on KDD, t_{CSS}+Annoy is the best-performing method overall. The performance drops on Netflix and GloVe for the similar reasons as above.

The t_{NNS} transformation. PCA-Trees make use of the t_{NNS} transformation. We found that t_{NNS}+PCA-Trees were among the slowest methods. For small depths (large leaves), recall was high but speedup limited because PCA-Trees do not perform pruning within leaves. For larger depths, recall dropped and speedup improved, particularly on the KDD dataset.

LEMP-LSHA. LEMP-LSHA satisfied the recall bound in all cases, and often gave better results than guaranteed. The left-most points in the figures corresponds to $R = 0.1$, the right-most points to $R = 0.9$. For $R = 0.9$, we obtained recall 0.99 on Netflix and 0.96 on KDD. One reason for this behavior is that LEMP-LSHA uses the exact NORM method on buckets where it considers LSH to be too expensive; this increases recall. LEMP-LSHA performed equally well with respect to ARE, although no guarantees are provided.

In our experiments, LEMP-LSHA performed equally well with t_{CSS} and the LSH-based FALCONN method on KDD, and outperformed it on Netflix and GloVe. One reason for this behavior is that LEMP-LSHA is more lightweight with respect to preprocessing: LSH signatures are created as needed for each bucket. Moreover, the untransformed probe vectors used by LEMP are less clustered: the average cosine similarity on a random sample was ≈ 0.1 for GloVe, ≈ 0.6 for Netflix, ≈ 0.3 for KDD. This makes LSHA more effective. LEMP-LSHA is currently based on random hyperplanes and it does not use multi-probing, whereas FALCONN uses cross-polytope LSH and multiprobing. An integration of FALCONN into LEMP along the lines of LSHA is a promising direction for further exploration.

LEMP-REL, LEMP-HYB-REL. LEMP-REL and LEMP-HYB-REL were the best-performing methods on Netflix and GloVe. On GloVe, LEMP-REL had up to 2x better recall than the closest non-LEMP method (t_{CSS}+Annoy) for the same computational cost, and was up to roughly 4x faster for the same recall level.

We observed that LEMP-REL can be orders of magnitude faster than LEMP, i.e., threshold augmentation was extremely beneficial. To understand why, we found that the vast majority of the true top-k results were found in roughly the first 25% of the probe buckets (largest L2 norms). At the time LEMP had processed all relevant buckets to retrieve the correct top-k list, it entered a "plateau" in the L2 norm distribution of the probe vectors. For this reason, it did not terminate and continued to process (many) further buckets. We observed the same behavior with LEMP-LSHA, which also was not able to prune these "excessive" buckets. In such a situation, threshold augmentation will lead to early termination without large losses in result quality. This observations also explains why a combination of threshold augmentation together with LSHA, like LEMP-HYB-REL, performed well.

On GloVe, LEMP-REL outperformed LEMP-HYB-REL for recall levels below 0.9. This is largely because the NI bucket-search method has significantly smaller preprocessing cost then the LSHA method used by LEMP-HYB-REL. On Netflix, this effect is less pronounced due to its lower dimensionality.

On KDD, the situation was the other way around: LEMP-REL was among the slowest methods, whereas LEMP-HYB-REL was second only to t_{CSS}+Annoy. The drop of performance in LEMP-REL results seems to stem from a more difficult coordinate distribution for the ICOORD bucket-search method. When analyzing the decisions of our tuner on the buckets that contained 75% of the results for $\epsilon = 0.4$, we observed that ICOORD scanned on average 7 coordinates (out of 51) on KDD. On GloVe, in comparison, it often only scanned a single list (out of 100). Thus an LSH-based bucket-search algorithm seems to be a better choice for KDD.

LEMP-LSHA(B), LEMP-HYB(B)-REL. We observed that using BayesLSH-Lite with LEMP-LSHA and LEMP-HYB-REL did not improve performance on Netflix and KDD. The reason is that the cost of BayesLSH advanced pruning techniques offset their benefits when inner products are cheap (small dimensionality, here $r = 50$). Notice that LEMP-HYB(B)-REL was slower than LEMP-HYB-REL on KDD, possibly because early termination eliminated the buckets where BayesLSH-Lite would have been most effective. On the other hand, vectors have higher dimensionality on GloVe ($r = 100$), and the pruning of BayesLSH-Lite started to pay off (up to 1.6x faster).

Discussion. To summarize, LEMP-REL and LEMP-HYB-REL were the best-performing methods on two out of the three datasets, whereas t_{CSS}+Annoy was the best-performing for the third one. This indicates that there is no single algorithm that works best in all settings.

8.5. Relative Performance of Bucket Algorithms

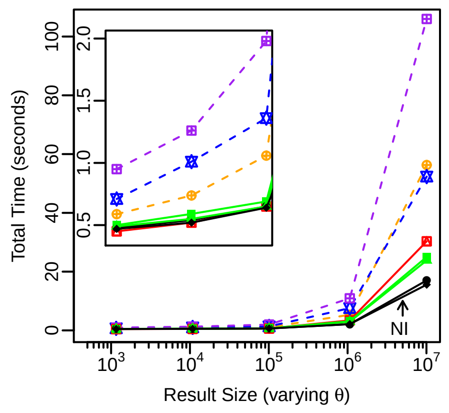
In the preceding experiments, we used NI as the bucket algorithm for LEMP because it provided the best overall performance. In this section, we consider and compare various alternative choices. Our results for exact MIPS are summarized in Figure 10. Wall-clock times for all experiments and average candidate set sizes can be found in Tables VI, VII and VIII in the online Appendix B (accessible in the ACM Digital Library). In the following, we discuss the performance of each algorithm in turn.

LEMP-N. For the IE datasets, LEMP-N was able to reduce the average candidate set size around 98% (13211 candidates per query vs. 771611 for Naive, IE-SVD^T, $k = 50$), whereas for datasets with less L2 norm skew the reduction ranged between 40% and 64% (Netflix) and 14% and 24% (KDD). Overall, LEMP-N was able to provide significant speedup over Naive: up to 17000x (670x) for IE-SVD and 15900x (440x) for IE-NMF for Above-θ-MIPS (Top-k-MIPS). In fact, the simple LEMP-N method outperformed all other methods for the IE datasets and small result sizes. I.e., bucket pruning was very effective for the datasets with large L2 norm skew. This indicates that LEMP's separate treatment of short and long vectors is beneficial. The performance of LEMP-N acts as a baseline for the performance of other bucket algorithms: LEMP-N's main filtering mechanism is bucket-level pruning, which is common to all LEMP methods.

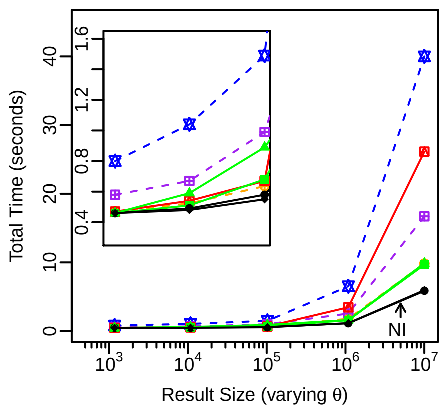
LEMP-C, LEMP-I. COORD created up to 7x less candidates per query than LEMP-N (e.g., 271 vs. 1915 for IE-NMF^T, $k = 1$) and its speedup over LEMP-N ranged between 2.7x and 4.7x. ICOORD reduced the candidates even further (34 candidates for IE-NMF^T, $k = 1$, 46x less than LEMP-N), with up to 7x speedup. The difference in the pruning power of COORD and ICOORD was more prevalent in the case of the KDD dataset (145K vs. 377K candidates per query). In the absence of large L2 norm skew or sparsity, ICOORD accumulates as much information as possible for the probe vectors. COORD, on the other hand, is not able to take full advantage of all the available information. For this reason, ICOORD was the best performing method (when LEMP was used together with only 1 bucket-algorithm) in terms of running time for the majority of datasets and configurations.

LEMP-NI. As discussed above, LEMP-N was the best performing method for datasets with high L2 norm skew on small retrieval levels. On the other hand, LEMP-I showed superior behavior in all other cases. LEMP-NI, for a small extra tuning cost, combines the strong points of both methods. In the majority of cases, it was the fastest method overall. In the remaining cases, the performance of LEMP-NI was similar to that of the best-performing method.

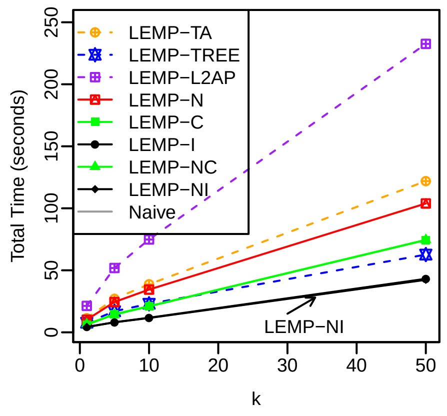
LEMP-TA. LEMP-TA was also able to offer speedup over LEMP-N for the sparse datasets: up to 3.5x for the Above-θ-MIPS (@ retrieval level 10M) and up to 6x for Top-



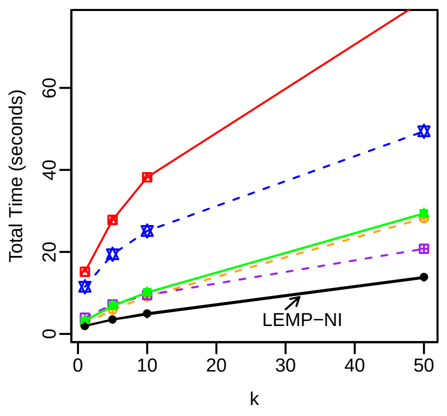
(a) Above-θ-MIPS IE-SVD



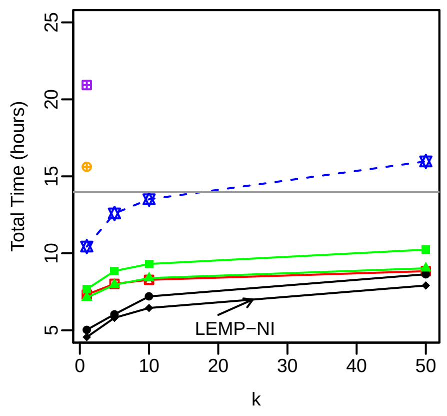
(b) Above-θ-MIPS IE-NMF



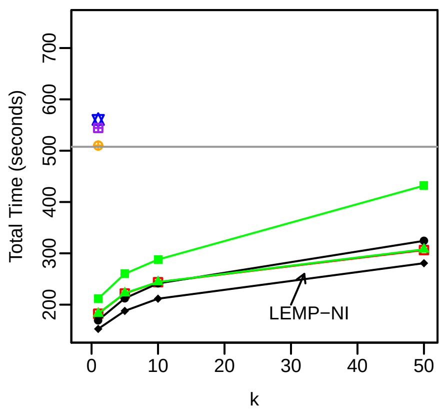
(c) Top-k-MIPS IE-SVD^T



(d) Top-k-MIPS IE-NMF^T



(e) Top-k-MIPS KDD



(f) Top-k-MIPS Netflix

Fig. 10: Comparison of LEMP bucket-algorithms in terms of total wall-clock times (incl. indexing and tuning) for exact MIPS

Table III: Performance (in terms of total wall-clock times) of LEMP-NI with and without using SIMD instructions on the KDD dataset for exact Top-k-MIPS. Time in minutes.

	$k = 1$	$k = 5$	$k = 10$	$k = 50$
No SIMD	273.3	348.8	386.9	474.5
SIMD-verify	239.6	294.5	318.6	383.7
SIMD-verify&scan	230.9	283.5	309.4	364.2

1-MIPS. However, it was usually outperformed by COORD and ICOORD: e.g., LEMP-I was up to 3x faster. The reason for ICOORD's superior behavior is that TA is usually not possible to identify good candidates by observing the value of only one coordinate. ICOORD avoids this problem by gathering information about the vectors from multiple coordinates (lists); based on this information, it prunes as many candidates as possible before actually calculating an inner product. Also note that LEMP-TA was significantly faster (up to 17x for IE-SVD^T, $k = 50$) than the standard TA algorithm, since the L2 norm-obliviousness and cache-misses problems are addressed by LEMP. This indicates that a method like LEMP might improve the performance of TA when linear scoring functions are used.

LEMP-L2AP. LEMP-L2AP was the method with the most aggressive pruning for all datasets (e.g., only 18 candidates per query for KDD, $k = 1$). However, this extensive pruning has a high cost: L2AP scans all the lists in the index that correspond to non-zero query coordinates and checks the filtering conditions during and after scanning. Also, the actual threshold used when querying the index can be far away from the lower bound used during index creation, which affects scanning time. For these reasons, ICOORD consistently outperformed L2AP (1.3x to 6.2x faster). Actually, L2AP was slower than Naive for both Netflix and KDD.

LEMP-Tree. LEMP-Tree creates one tree per bucket (lazy construction), instead of one tree for the entire probe dataset. This explains why LEMP-Tree had much better performance than Tree (up to 10x faster) for the datasets for which preprocessing was Tree's bottleneck (see Above-θ-MIPS experiments, small result sizes). In terms of pruning power, LEMP-Tree did not have a consistent behavior w.r.t. Tree. For datasets with large L2 norm skew (IE-NMF^T, IE-SVD^T), LEMP-Tree checked less candidates per query, whereas for datasets for small skew (e.g., KDD) it checked more. However, even in these cases, LEMP-Tree was faster than Tree, due to the better cache utilization provided by the bucketization.

8.6. Parallel LEMP

In our final set of experiments, we investigated the performance and scalability of our parallel versions of LEMP.

Instruction-level parallelism. Table III shows the performance of LEMP-NI on our largest dataset (KDD) without SIMD, with SIMD in verification, and with SIMD in verification and scanning. We observed that for large values of k, for which less opportunities for pruning exist (low value of $\hat{\theta}$) and more candidates need to be verified, the speedup due to SIMD in verification reaches 19%. For small values of k, the speedup is lower (12% for $k = 1$). Using instruction-level-parallelism for scanning in addition, further improved the runtime by 6%.

Multi-threading. Figure 11 shows the performance of LEMP-NI in terms of wall-clock time for a number of processors varying between 1 and 32. For each setting, we also give the speedup compared to sequential processing. The figure shows results for Top-1-MIPS on the KDD dataset on an Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz

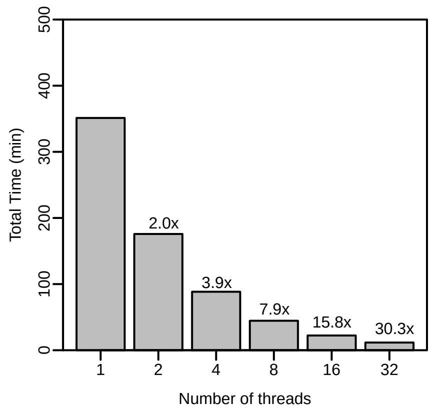


Fig. 11: Scalability of LEMP-NI for KDD, exact Top-1-MIPS

with 40 cores and 512GB RAM. Note that even for embarrassingly parallel problems such as multi-query MIPS, linear speedups are rarely achieved when a large number of processors is used. This is mainly because memory bandwidth and synchronization quickly become a bottleneck. Figure 11 shows that LEMP was able to achieve near linear speedups even for large numbers of processors. This indicates that LEMP's careful cache utilization and avoidance of synchronization is effective.

9. Conclusion

We proposed LEMP, a novel framework for exact and approximate MIPS. At its heart, LEMP bucketizes probe vectors by their L2 norm. During query processing, LEMP prunes buckets and selects suitable search strategies for the remaining buckets. Buckets are small, so that they fit into the cache, and indexed in a light-weight, data-dependent, and lazy way.

For exact MIPS, LEMP prunes buckets only when they cannot contribute to the result and uses exact search strategies within buckets. We proposed two novel search strategies—COORD and ICOORD—that worked particularly well in LEMP's setting. For approximate MIPS, LEMP may also prune "unpromising" buckets and/or use approximate search within buckets. We proposed multiple approaches to do so, each providing different quality guarantees and speed-quality trade-offs.

We investigated the performance of many state-of-the-art approaches and found that LEMP-based methods consistently (and with only one exception) provided the best overall performance.

REFERENCES

Thomas D. Ahle, Rasmus Pagh, Ilya Razenshteyn, and Francesco Silvestri. 2016. On the Complexity of Inner Product Similarity Join *(PODS '16)*. ACM.

David C. Anastasiu and George Karypis. 2014. L2AP: Fast Cosine Similarity Search With Prefix L-2 Norm Bounds. In *ICDE*. 12.

Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *NIPS*.

Yoram Bachrach, Yehuda Finkelstein, Ran Gilad-Bachrach, Liran Katzir, Noam Koenigstein, Nir Nice, and Ulrich Paquet. 2014. Speeding Up the Xbox Recommender System Using a Euclidean Transformation for Inner-product Spaces. In *RecSys*. 257–264.

Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling Up All Pairs Similarity Search. In *WWW*. 131–140. DOI:http://dx.doi.org/10.1145/1242572.1242591

James Bennett and Stan Lanning. 2007. The Netflix Prize. In *KDD Cup and Workshop*.

Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover Trees for Nearest Neighbor. In *ICML*. 97–104. DOI:http://dx.doi.org/10.1145/1143844.1143857

Moses S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing (STOC '02)*. ACM, New York, NY, USA, 380–388. DOI:http://dx.doi.org/10.1145/509907.509965

Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*.

Ryan R. Curtin and Parikshit Ram. 2014. Dual-tree fast exact max-kernel search. *Statistical Analysis and Data Mining* (2014), 229–253. DOI:http://dx.doi.org/10.1002/sam.11218

Ryan R. Curtin, Parikshit Ram, and Alexander G. Gray. 2013. Fast Exact Max-Kernel Search. In *SDM*. 1–9.

Thomas Dean, Mark Ruzon, Mark Segal, Jonathon Shlens, Sudheendra Vijayanarasimhan, and Jay Yagnik. 2013. Fast, Accurate Detection of 100,000 Object Classes on a Single Machine. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA.

Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. 2012. The Yahoo! Music Dataset and KDD-Cup '11. *Journal of Machine Learning Research: Proceedings Track* (2012).

Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. In *PODS*. 102–113.

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity Search in High Dimensions via Hashing. In *VLDB*. 518–529.

Noam Koenigstein, Gideon Dror, and Yehuda Koren. 2011. Yahoo! Music Recommendations: Modeling Music Ratings with Temporal Dynamics and Item Taxonomy. In *RecSys*. 165–172.

Noam Koenigstein, Parikshit Ram, and Yuval Shavitt. 2012. Efficient retrieval of recommendations in a matrix factorization framework. In *CIKM*. 535–544.

Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *IEEE Computer* 42, 8 (2009), 30–37.

Dongjoo Lee, Jaehui Park, Junho Shim, and Sang-goo Lee. 2010. An Efficient Similarity Join Algorithm with Cosine Similarity Predicate. In *DEXA: Part II*. 422–436.

Faraz Makari, Christina Teflioudi, Rainer Gemulla, Peter Haas, and Yannis Sismanis. 2015. Shared-memory and shared-nothing stochastic gradient descent algorithms for matrix completion. *Knowledge and Information Systems* (2015).

Ndapandula Nakashole, Gerhard Weikum, and Fabian Suchanek. 2012. PATTY: A Taxonomy of Relational Patterns with Semantic Types. In *EMNLP-CoNLL*. 1135–1145.

B. Neyshabur and N. Srebro. 2014. On Symmetric and Asymmetric LSHs for Inner Product Search. *ArXiv e-prints* (Oct. 2014).

Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. 2011. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Parikshit Ram and Alexander G. Gray. 2012. Maximum inner-product search using cone trees. In *KDD*. 931–939.

Benjamin Recht and Christopher Ré. 2011. Parallel Stochastic Gradient Algorithms for Large-Scale Matrix Completion. *Optimization Online* (2011).

Sebastian Riedel, Limin Yao, Benjamin M. Marlin, and Andrew McCallum. 2013. Relation Extraction with Matrix Factorization and Universal Schemas. In *HLT-NAACL*.

Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *Proc. VLDB Endow.* 5, 5 (Jan. 2012), 430–441. DOI:http://dx.doi.org/10.14778/2140436.2140440

Anshumali Shrivastava and Ping Li. 2014a. Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *NIPS*. 2321–2329.

Anshumali Shrivastava and Ping Li. 2014b. An Improved Scheme for Asymmetric LSH. *CoRR* abs/1410.5410 (2014).

David Skillicorn. 2007. *Understanding complex datasets: data mining with matrix decompositions*. Taylor & Francis Ltd.

Christina Teflioudi, Rainer Gemulla, and Olga Mykytiuk. 2015. LEMP: Fast Retrieval of Large Entries in a Matrix Product *(SIGMOD '15)*. ACM.

Christina Teflioudi, Faraz Makari, and Rainer Gemulla. 2012. Distributed Matrix Completion. In *ICDM*.

Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Efficient Similarity Joins for Near Duplicate Detection. In *WWW*. 131–140. DOI:http://dx.doi.org/10.1145/1367497.1367516

Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient Similarity Joins for Near-duplicate Detection. *ACM Trans. Database Syst.* 36, 3, Article 15 (Aug. 2011), 41 pages. DOI:http://dx.doi.org/10.1145/2000824.2000825

Zhiwei Zhang, Qifan Wang, Lingyun Ruan, and Luo Si. 2014. Preference Preserving Hashing for Efficient Recommendation. In *SIGIR*. 183–192. DOI:http://dx.doi.org/10.1145/2600428.2609578

A. Implementation Details

In this section, we give some guidance on how to implement the COORD, ICOORD, and LSHA algorithms efficiently.

COORD. In our implementation, we store the sorted-list indexes column-wise to reduce memory bandwidth: the data values are accessed only during binary search to determine the scan range, and the local identifiers are accessed only during the actual scan phase. For efficiency reasons, we also avoid clearing the CP array when moving from one query vector to the next. Instead, we keep the array uninitialized and proceed as follows. When scanning the first sorted list, we set to 1 instead of incrementing the corresponding entry of the CP array and increment while scanning the remaining sorted lists. After all lists have been scanned, we scan the first sorted list again and only consider the corresponding entries of the CP array for inclusion into the candidate set. Since the first sorted list is scanned twice (for CP array initialization and filtering), we take the focus coordinate with the smallest scan range as the first one.

ICOORD. Since ICOORD needs access to both coordinate values and local identifiers during scanning, we store the sorted lists row-wise. The extended CP array is initialized and accessed in the same way as the CP array of COORD. In order to reduce memory bandwidth and avoid excessive checking, we do not keep the counter information of COORD in the extended CP array: the filtering condition of Eq. (13) is usually pruning vectors more aggressively than the simple check of COORD. Since Eq. (13) contains expensive floating-point operations (such as divisions and square roots), we rewrite the conditions and accept a vector $\bar{\boldsymbol{p}}$ if

$$\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F \|\boldsymbol{p}\| > \theta / \|\boldsymbol{q}\|,$$

for which the right-hand side needs to be computed only once. If this test fails, we accept $\bar{\boldsymbol{p}}$ if and only if

$$\|\boldsymbol{p}\|^2 \|\boldsymbol{q}\|^2 (1 - \|\bar{\boldsymbol{p}}_F\|^2)(1 - \|\bar{\boldsymbol{q}}_F\|^2) \geq (\theta - \bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F \|\boldsymbol{p}\| \|\boldsymbol{q}\|)^2.$$

ICOORD's strength lies in accumulating partial inner products from many lists. If we decide to use $\phi_b = 1$ for some bucket b, ICOORD and COORD will produce the same candidate set, but COORD does so faster. We thus use COORD instead of ICOORD whenever $\phi_b = 1$.

LSHA. To create the random vectors $\boldsymbol{u}$ from the standard Normal distribution, we follow the approach of Satuluri and Parthasarathy [2012], which allows for compressed storage. In addition, using Eq. (15) for each query-bucket pair can be expensive. In our implementation, we precompute and cache the smallest local threshold that corresponds to each one the 200 signatures in the budget. During query processing, we perform binary search on these values to find $\mathcal{L}$ for each $\theta_b(\boldsymbol{q})$.

B. Additional Experiments

Tables IV - VIII show running times for exact Above-θ-MIPS and Top-k-MIPS experiments for different retrieval levels and values of k (including those presented in the figures of Sec. 8).

Table IV: Comparison of LEMP with state-of-the-art algorithms for the exact Above-θ-MIPS problem w.r.t. wall-clock time (in seconds). We give the average candidate set size per query in parentheses. The last column gives the preprocessing time as minimum/maximum percentage of the overall time (occurs at large/small result sizes, resp.).

Dataset	Algorithm	@1K		@10K		@100K		@1M		@10M		Preprocessing
		Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	time [%]
IE-SVD	Naive	6818.8	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-
	Tree	7.74	*(2.1)*	7.96	*(3.1)*	8.36	*(4.9)*	12.67	*(30.8)*	50.74	*(236.8)*	15% – 95%
	D-Tree	46.33	***(0.2)***	46.36	***(0.4)***	46.47	***(0.7)***	47.30	***(5.9)***	57.94	***(66.2)***	80% – 99%
	TA	2.31	*(3.1)*	2.71	*(6.9)*	3.30	*(12.6)*	10.92	*(81.54)*	98.84	*(818.8)*	<1% – 32%
	LEMP-NI	**0.41**	*(1.03)*	**0.47**	*(0.44)*	**0.56**	*(1.06)*	**2.07**	*(17.47)*	**16.20**	*(217.49)*	8% – 90%
IE-NMF	Naive	6818.8	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-
	Tree	5.89	*(2.3)*	6.10	*(3.3)*	6.50	*(5.2)*	10.24	*(29.4)*	39.41	*(185.4)*	14% – 93%
	D-Tree	38.60	*(0.2)*	38.62	***(0.3)***	38.72	***(0.7)***	39.46	***(5.1)***	47.06	***(46.6)***	81% – 99%
	TA	1.91	*(0.9)*	1.93	*(1.2)*	2.02	*(2.1)*	2.91	*(11.4)*	17.50	*(139.8)*	2.5% –23%
	LEMP-NI	**0.39**	***(0.1)***	**0.44**	*(1.14)*	**0.58**	*(2.4)*	**1.15**	*(7.24)*	**6.11**	*(51.03)*	14% – 88%

Table V: Comparison of LEMP with state-of-the-art algorithms for the exact Top-k-MIPS problem w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses. The last column gives the preprocessing time as minimum/maximum percentage of the overall time (occurs at large/small k, resp.).

Dataset	Algorithm	k=1		k=5		k=10		k=50		Preprocessing
		Time	$\vert C\vert/q$	Time	$\vert C\vert/q$	Time	$\vert C\vert/q$	Time	$\vert C\vert/q$	time [%]
IE-SVD^T	Naive	6818.8	*(771K)*	-	*(771K)*	-	*(771K)*	-	*(771K)*	-
	Tree	50.6	*(357)*	63.7	*(772)*	75.1	*(1119)*	158.5	*(3213)*	25% – 77%
	D-Tree	4766.8	*(24K)*	5826.8	*(29K)*	6285.1	*(30K)*	8408.5	*(34K)*	<1%
	TA	200.60	*(6090)*	644.71	*(19482)*	936.62	*(28036)*	2100.08	*(62037)*	<1% –2.4%
	LEMP-NI	**4.43**	***(54)***	**8.05**	***(285)***	**11.55**	***(512)***	**42.32**	***(1772)***	3.7% – 15.4%
IE-NMF^T	Naive	6818.8	*(771K)*	-	*(771K)*	-	*(771K)*	-	*(771K)*	-
	Tree	46.8	*(465)*	58.2	*(845)*	67.3	*(1107)*	122.5	*(2591)*	27% – 70%
	D-Tree	3169.8	*(16K)*	3642.6	*(18K)*	3915.2	*(19K)*	5203.1	*(21K)*	<1% –1.2%
	TA	55.15	*(1899)*	169.04	*(5514)*	232.10	*(7552)*	455.99	*(14721)*	<1% –5.4%
	LEMP-NI	**1.95**	***(33)***	**3.48**	***(100)***	**4.76**	***(136)***	**13.66**	***(417)***	7% – 23%
Netflix	Naive	507.6	*(17.7K)*	-	*(17.7K)*	-	*(17.7K)*	-	*(17.7K)*	-
	Tree	1289.1	*(9279)*	>Naive	-	>Naive	-	>Naive	-	<1%
	D-Tree	6095.6	*(11K)*	>Naive	-	>Naive	-	>Naive	-	<57%
	TA	1961.8	*(16K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	LEMP-NI	**152.72**	***(4108)***	**187.78**	***(4485)***	**211.59**	***(5889)***	**280.74**	***(8151)***	<1%
KDD	Naive	14h	*(624K)*	-	*(624K)*	-	*(624K)*	-	*(624K)*	-
	Tree	15.2h	*(86K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	D-Tree	15.1h	***(72K)***	>Naive	-	>Naive	-	>Naive	-	<6
	TA	44.7h	*(567K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	LEMP-NI	**4.56h**	*(85K)*	**5.81h**	***(149K)***	**6.45h**	***(187K)***	**7.91h**	***(277K)***	<1%
GloVe	Naive	290.4m	*(1093K)*	-		-		-		-
	Tree	1305.4m	*(882K)*	>Naive	-	>Naive	-	>Naive	-	<45.8%
	D-Tree	1360.7m	*(856K)*	>Naive	-	>Naive	-	>Naive	-	<44.5
	TA	880.5m	*(1093K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	LEMP-NI	**128.7m**	***(482K)***	**181.6m**	***(674K)***	**197.6m**	***(838K)***	**228.2m**	***(1028K)***	<1%

Table VI: Comparison of LEMP bucket algorithms for the exact Above-θ-MIPS problem w.r.t. wall-clock time (in seconds). We give the average candidate set size per query in parentheses.

Dataset	Algorithm	@1K		@10K		@100K		@1M		@10M	
		Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$
IE-SVD	LEMP-L2AP	1.02	***(0.09)***	1.33	***(0.14)***	2.07	***(0.39)***	11.38	***(1.49)***	107.69	***(14.16)***
	LEMP-C	0.65	*(1.61)*	0.52	*(2.89)*	0.62	*(4.66)*	2.87	*(43.37)*	23.08	*(432.21)*
	LEMP-I	0.43	*(0.18)*	0.47	*(0.42)*	0.57	*(1.06)*	2.14	*(16.18)*	17.18	*(227.36)*
	LEMP-N	**0.40**	*(1.21)*	**0.46**	*(2.64)*	0.61	*(5.47)*	3.61	*(71.36)*	32.86	*(716.15)*
	LEMP-NC	0.41	*(1.09)*	0.48	*(2.15)*	0.59	*(3.8)*	2.78	*(42.73)*	22.45	*(431.96)*
	LEMP-NI	0.41	*(1.03)*	0.47	*(0.44)*	**0.56**	*(1.06)*	**2.07**	*(17.47)*	**16.20**	*(217.49)*
	LEMP-TA	0.55	*(1.84)*	0.80	*(2.98)*	1.01	*(5.35)*	5.27	*(30.74)*	55.60	*(453.64)*
	LEMP-TREE	0.67	*(1.71)*	0.92	*(3.14)*	1.39	*(5.8)*	7.79	*(46.58)*	57.01	*(320.44)*
IE-NMF	LEMP-L2AP	0.77	*(0.1)*	0.84	***(0.17)***	1.03	*(0.44)*	2.53	***(1.51)***	15.96	***(14.11)***
	LEMP-C	0.42	*(0.75)*	0.45	*(1.21)*	0.56	*(2.59)*	1.46	*(15.92)*	9.44	*(147.85)*
	LEMP-I	0.40	***(0.09)***	**0.44**	***(0.17)***	**0.50**	***(0.43)***	**1.10**	*(4.26)*	**5.92**	*(44.38)*
	LEMP-N	0.42	*(1.5)*	0.48	*(2.88)*	0.62	*(5.76)*	3.55	*(70.02)*	34.29	*(614.01)*
	LEMP-NC	0.42	*(0.77)*	0.44	*(1.23)*	0.51	*(2.6)*	1.39	*(15.96)*	8.93	*(148.67)*
	LEMP-NI	**0.39**	*(0.1)*	0.44	*(1.14)*	0.58	*(2.4)*	1.15	*(7.24)*	6.11	*(51.03)*
	LEMP-TA	0.43	*(0.55)*	0.47	*(0.85)*	0.59	*(1.73)*	1.65	*(4.94)*	9.72	*(62.34)*
	LEMP-TREE	0.78	*(2.7)*	1.04	*(4.37)*	1.51	*(7.55)*	6.82	*(43.58)*	43.63	*(251.14)*

Table VII: Comparison of LEMP bucket algorithms for the exact Top-k-MIPS problem on the IE datasets w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses.

Dataset	Algorithm	k=1		k=5		k=10		k=50	
		Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$
IE-SVD^T	LEMP-L2AP	21.30	***(15)***	51.84	***(24)***	74.86	***(35)***	232.57	***(129)***
	LEMP-C	6.06	*(541)*	14.63	*(1418)*	20.99	*(2182)*	74.41	*(7823)*
	LEMP-I	**4.17**	*(61)*	8.13	*(261)*	11.76	*(565)*	43.18	*(2325.19)*
	LEMP-N	10.16	*(1272)*	24.44	*(3102)*	34.47	*(4386)*	103.90	*(13211)*
	LEMP-NC	6.68	*(685)*	14.23	*(1436)*	20.97	*(2190)*	74.52	*(7803)*
	LEMP-NI	4.43	*(54)*	**8.05**	*(285)*	**11.55**	*(512)*	**42.32**	*(1772)*
	LEMP-TA	11.41	*(578)*	27.04	*(1448)*	38.75	*(2092)*	121.91	*(6521)*
	LEMP-TREE	7.75	*(213)*	16.95	*(536)*	23.00	*(744)*	62.67	*(2034)*
IE-NMF^T	LEMP-L2AP	3.96	***(11)***	7.21	***(24)***	9.48	***(33)***	20.76	***(126)***
	LEMP-C	3.20	*(271)*	6.83	*(659)*	10.11	*(1001)*	29.40	*(3039)*
	LEMP-I	1.97	*(34)*	3.53	*(89)*	5.02	*(146)*	13.89	*(343)*
	LEMP-N	15.16	*(1915)*	27.81	*(3541)*	38.20	*(4869)*	81.39	*(10319)*
	LEMP-NC	3.20	*(274)*	6.90	*(664)*	10.15	*(1026)*	29.36	*(3041)*
	LEMP-NI	**1.95**	*(33)*	**3.48**	*(100)*	**4.76**	*(136)*	**13.66**	*(417)*
	LEMP-TA	2.35	*(89)*	5.93	*(328)*	9.12	*(540)*	28.23	*(1757)*
	LEMP-TREE	11.50	*(345)*	19.43	*(640)*	25.11	*(851)*	49.42	*(1693)*

Table VIII: Comparison of LEMP bucket algorithms for the exact Top-k-MIPS problem on KDD, Netflix w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses.

Dataset	Algorithm	k=1		k=5		k=10		k=50	
		Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$	Time	$\lvert C\rvert/q$
KDD	LEMP-L2AP	20.93h	***(18)***	>Naive	-	>Naive	-	>Naive	-
	LEMP-C	7.67h	*(377K)*	8.85h	*(437K)*	9.31h	*(461K)*	10.24h	*(507K)*
	LEMP-I	5.03h	*(145K)*	6.04h	*(175K)*	7.20h	*(281K)*	8.64h	*(366K)*
	LEMP-N	7.32h	*(445K)*	8.02h	*(488K)*	8.28h	*(504K)*	8.84h	*(537K)*
	LEMP-NC	7.12h	*(375K)*	7.96h	*(437K)*	8.39h	*(460K)*	9.03h	*(506K)*
	LEMP-NI	**4.56**	*(85K)*	**5.81h**	*(149K)*	**6.45h**	*(187K)*	**7.91h**	*(277K)*
	LEMP-TA	15.62h	*(412K)*	>Naive	-	>Naive	-	>Naive	-
	LEMP-TREE	10.44h	*(196K)*	12.59h	*(237K)*	13.51h	*(258K)*	15.97h	*(307K)*
Netflix	LEMP-L2AP	544.30	***(32)***	>Naive	-	>Naive	-	>Naive	-
	LEMP-C	211.59	*(6112)*	260.11	*(7465)*	287.93	*(8242)*	431.95	*(10291)*
	LEMP-I	169.57	*(3908)*	212.29	*(4601)*	241.39	***(5709)***	324.31	***(8120)***
	LEMP-N	182.39	*(6486)*	221.81	*(7839)*	243.82	*(8593)*	306.43	*(10587)*
	LEMP-NC	182.70	*(6284)*	222.37	*(7688)*	244.12	*(8483)*	307.74	*(10565)*
	LEMP-NI	**152.72**	*(4108)*	**187.78**	***(4485)***	**211.59**	*(5889)*	**280.74**	*(8151)*
	LEMP-TA	510.04	*(7284)*	>Naive	-	>Naive	-	>Naive	-
	LEMP-TREE	559.97	*(6475)*	>Naive	-	>Naive	-	>Naive	-